Mastering Python For Data Science

Authored by :
C.itp Clive Asuai, Houssem HOSNI
& Muhammad Ibrahim Khan

Copyright

Mastering Python for Data Science
© 2025 Clive et al

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the author, except in the case of brief quotations embodied in critical articles or reviews.

First Edition
Published in Nigeria

ISBN: 978-1-234567-89-0
Cover design by Brainiacdesigns
For permissions, inquiries or feedback, contact:
cliveebomagune@gmail.com

Printed and Published by
Glit Publishers
Plot 211 Mahuta New Extension
By Yakowa Express Way Kaduna.
Glitworkspaces@gmail.com
Tel: +2348026978666

CONTENTS

DEDICATION **xiii**
ACKNOWLEDGEMENT **xiv**
PREFACE **xv**

MODULE 1: FOUNDATIONS OF DATA SCIENCE AND PYTHON PROGRAMMING **1**

- Introduction to Data Science
- Programming Languages in Data Science
- Python for Data Science
- Development Environments
- Getting Started with Python
- Python Programming Basics
- Review Questions

MODULE 2: GETTING STARTED WITH PYTHON **33**

- Introduction to Python Basics
- Variables in Python
- Data Types in Python
- Type Conversion and Casting
- Working with Dates and Times
- Operators in Python
- Functions in Python
- Functions as Objects
- Object-Oriented Concepts in Python
- Practical Applications in Data Science
- Review Questions

MODULE 3: CONTROL STRUCTURES 77

- Introduction to Control Structures
- Overview of Python control structures
- Conditional Statements (Decision Making)
- if statements
- if-else statements
- if-elif-else statements
- Nested conditionals
- Logical operators (and, or, not) in conditions
- Looping Constructs (Iteration)
- for loops (iterating over sequences)
- while loops (condition-based iteration)
- Nested loops
- Loop Control Statements
- break (exiting loops early)
- continue (skipping iterations)
- pass (placeholder for future code)
- The range() Function
- Ternary Expressions (Conditional Expressions)
- Applications in data science (e.g., labeling, feature engineering)
- Practice Questions

MODULE 4 INTRODUCTION TO PYTHON LIBRARIES FOR DATA SCIENCE 93

- Introduction to Python Libraries
- Importing Python Libraries
- Core Python Libraries for Data Science
- NumPy – Numerical Python
- pandas – Panel Data Analysis
- Matplotlib – Plotting and Visualization
- Seaborn – Statistical Data Visualization
- scikit-learn – Machine Learning Made Simple
- SciPy – Scientific Computing and Advanced Math
- Hands-On Exercises

MODULE 5: FILE HANDLING IN PYTHON FOR DATA SCIENCE 107

- Introduction
- Basics of File Handling in Python
- Reading Files
- Writing and Appending to Files
- Working with CSV Files
- Handling JSON Files
- Best Practices for Closing Files
- Parsing and Extracting Structured Data
- Handling Missing Values in Files Using pandas
- Integrating File Handling with the Data Science Ecosystem
- Review Questions

MODULE 6: DATA STRUCTURES IN PYTHON 125

- Introduction to Data Structures
- Lists
- Tuples
- Sets
- Dictionaries
- Comprehensions
- Performance and Mutability
- Practical Applications in Data Science
- Review Questions

MODULE 7: DATA MANIPULATION AND ANALYSIS WITH NUMPY AND PANDAS 157

- Introduction
- Working with NumPy
- Data Handling with Pandas
- Data Manipulation with Pandas
- Data Analysis with NumPy and Pandas

- Basic Data Visualization with Pandas
- Applying NumPy and Pandas to Real-world Scenarios
- Exercises and Practice Tasks
- Mini Project: Analyzing Real-world Dataset Using NumPy and Pandas

MODULE 8: DATA VISUALIZATION **171**

- Introduction
- Data Visualization with Matplotlib
- Advanced Matplotlib Techniques
- Data Visualization with Seaborn
- Customizing Seaborn Visualizations
- Real-World Applications
- Review Questions

MODULE 9: LINEAR ALGEBRA FOR DATA SCIENCE **203**

- Introduction to Linear Algebra
- Vectors and Vector Operations
- Matrices and Matrix Operations
- Eigenvalues and Eigenvectors
- Singular Value Decomposition (SVD)
- Applications in Data Science
- Review Questions

MODULE 10: ADVANCED NUMPY – ARRAYS AND VECTORIZED COMPUTATION **227**

- Introduction to NumPy
- NumPy Arrays (ndarray)
- Array Operations
- Indexing and Slicing
- Reshaping and Transposing Arrays
- Mathematical and Statistical Methods

- Sorting and Set Operations
- Advanced NumPy Functions
- File I/O with NumPy
- Pseudorandom Number Generation
- Review Questions

MODULE 11: PROBABILITY AND STATISTICS 267

- Introduction to Probability and Statistics
- Probability Fundamentals
- Descriptive Statistics
- Inferential Statistics
- Applications in Data Science
- Case Study: Customer Spending Analysis
- Review Questions

MODULE 12: ADVANCED PANDAS FOR DATA SCIENCE 297

- Introduction to Advanced Pandas
- Advanced Indexing Techniques
- Efficient Data Manipulation
- Boolean Indexing & Querying
- Group Operations & Aggregations
- Window Functions
- Time Series Analysis
- Performance Optimization
- Case Studies & Practical Applications
- Review Questions

MODULE 13: ERRORS AND EXCEPTION HANDLING 353

- Introduction
- Basics of Errors and Exceptions in Python
- Types of Python Errors
- Handling Runtime Errors with try-except

- Python Built-in Exceptions
- Importance of Exception Handling in Data Science
- Basic Exception Handling Techniques
- Advanced Exception Handling
- Applying Exception Handling in Data Science
- Debugging and Logging Exceptions
- Best Practices for Exception Handling
- Practical Data Science Scenarios
- Writing Robust and Error-Resilient Code
- Exception Handling in Production Environments
- Review Questions

MODULE 14: PLOTTING AND VISUALIZATION 375

- Introduction to Data Visualization
- Types of Data and Visualizations
- Getting Started with Matplotlib
- Creating Basic Plots
- Customizing Plots in Matplotlib
- Subplots and Multi-Panel Figures
- Introduction to Seaborn
- Advanced Seaborn Features
- Quick Visualizations with Pandas
- Exploratory Data Analysis (EDA) with Visualization
- Interactive and Geospatial Visualization
- Advanced Topics
- Case Studies and Applications
- Review Questions

MODULE 15: TIME SERIES ANALYSIS 445

- Introduction to Time Series Data
- Working with Dates and Times in Python
- Handling Time Series in Pandas
- Managing Duplicate and Missing Timestamps

- Date Ranges and Frequencies
- Shifting and Resampling
- Rolling and Expanding Window Operations
- Time Zone Handling
- Advanced Time Series Operations
- Practical Applications
- Review Questions

MODULE 16: ADVANCED TIME SERIES ANALYSIS 463

- Introduction to Time Series Analysis
- Preprocessing Time Series Data
- Time Series Decomposition
- Stationarity and Testing
- Autocorrelation Analysis
- Time Series Forecasting Models
- Advanced Models
- Model Evaluation
- Real-World Applications
- Advanced Topics
- Review Questions

MODULE 17: MACHINE LEARNING WITH PYTHON 497

- Introduction to Machine Learning
- Supervised Learning
- Unsupervised Learning
- Performance Metrics in Machine Learning
- Classification Model Evaluation Metrics
- Regression Model Evaluation Metrics
- Practical Demonstrations and Model Comparison
- Review Questions

MODULE 18: REAL-WORLD DATA SCIENCE PROJECTS 517

- End-to-End Data Science Project
- Problem Definition
- Data Collection
- Data Cleaning
- Handling Missing Values, Duplicates, and Inconsistencies
- Model Building
- Feature Selection, Train-Test Split, Algorithm Selection
- Model Evaluation
- Deployment
- Saving the Model (Joblib/Pickle)
- Creating a RESTful API (Flask/FastAPI)
- Docker Containerization
- Cloud Deployment (Heroku, AWS, Azure)
- Feature Engineering in Data Science
- Steps in Feature Engineering
- Feature Creation, Transformation, Selection, Extraction
- Feature Selection
- Ensemble Feature Selection:
- 3ConFA Framework (Chi-Square + IG + DT-RFE)
- Predictive Analytics
- Customer Segmentation
- Sentiment Analysis
- Review Questions

MODULE 19: HANDS-ON PROJECTS FOR DATA SCIENCE FUNDAMENTALS AND BEYOND 543

- Project 1: Perform EDA on a Simple Dataset
- Project 2: Basic Linear Regression for House Price Prediction
- Project 3: Simple Classification with Logistic Regression
- Project 4: Basic Sentiment Analysis with TextBlob
- Project 5: Data Visualization of Weather Data

- Project 6: Real-World Data Exploration and Visualization
- Project 7: Build an Interactive Dashboard
- Project 8: Time Series Forecasting with ARIMA
- Project 9: Deep Learning for Image Classification
- Project 10: Natural Language Processing for Sentiment Analysis
- Project 11: Predictive Maintenance for Industrial Equipment
- Project 12: Fraud Detection in Financial Transactions
- Project 13: Recommender System for Movie Recommendations
- Project 14: Machine Translation with Neural Networks
- Project 15: Anomaly Detection in Cybersecurity
- Project 16: Predicting Disease Spread with Machine Learning
- Project 17: Customer Segmentation with K-means Clustering
- Project 18: Genetic Algorithm for Optimization Problems
- Project 19: Predicting Loan Default with Gradient Boosting
- Project 20: Market Basket Analysis with Association Rule Learning
- Project 21: Image Captioning with Deep Learning
- Project 22: Text Generation with Recurrent Neural Networks
- Project 23: Neural Style Transfer for Image Transformation
- Project 24: Predicting Housing Prices with Ensemble Methods
- Project 25: Building a Chatbot with NLP
- Project 26: Image Super-Resolution with Deep Learning
- Project 27: Social Media Sentiment and Trend Analysis
- Project 28: Multimodal Emotion Recognition from Text, Audio, and Video

Dedication

This book is dedicated to God almighty and family, friends, mentors, students. Your support, encouragement, and belief in us made this journey possible.

Acknowledgments

We would like to express our heartfelt appreciation to everyone who contributed to the creation of this book. Special thanks to our students, whose curiosity and feedback inspired many of the examples and case studies included. We are grateful to our colleagues and collaborators for their technical insights and encouragement throughout the writing process.
We also wish to acknowledge the authors of the following works, whose content and ideas greatly influenced this book:

- **Peter Morgan**, for his invaluable contributions in *Data Analysis from Scratch with Python: Step-by-Step Guide*, which provided a foundational understanding of Python in data analysis.
- **Laura Igual & Santi Seguí**, for their work in *Introduction to Data Science: A Python Approach to Concepts, Techniques, and Applications*, which greatly enriched the conceptual depth of the data science methods discussed.
- **Wes McKinney**, for his comprehensive work in *Python for Data Analysis (2nd Edition)*, which served as a key reference for practical Python applications in data analysis.

Thanks
Clive et al

Preface

Mastering Python for Data Science is designed as a comprehensive, hands-on guide for learners and professionals aiming to build solid, practical skills in Python programming and its applications in data science. This book blends core Python concepts with real-world data challenges, providing a clear pathway from fundamental programming techniques to advanced data analysis, modeling, and visualization.

Each module is structured around practical case studies and real-life problems drawn from various domains such as finance, health, social media, and business analytics. To reinforce learning, modules conclude with practical exercises, short answer questions, and coding tasks that challenge the reader to apply concepts immediately.

Purpose of the Book

The goal of this book is to provide a step-by-step, project-oriented approach to learning Python for data science. It covers essential topics such as data types, control structures, data wrangling with Pandas, data visualization with Matplotlib and Seaborn, and machine learning with scikit-learn.

By integrating theory with application, this book equips readers with the practical tools and problem-solving techniques needed to work confidently with real datasets. The inclusion of case studies and end-of-module assessments ensures that readers can test and deepen their understanding in a meaningful context

Target Audience

This book is intended for:

- Students studying data science, computer science, or analytics.
- Aspiring data scientists and analysts looking to gain practical Python skills.
- Professionals transitioning into data science roles.
- Anyone with a basic knowledge of programming who wants to apply Python to solve real-world data problems.

Prerequisites

Before using this book, readers should have a basic understanding of programming concepts such as variables, loops, and functions. Familiarity with Python syntax is helpful but not required, as the book provides a quick refresher in the early modules. A basic knowledge of mathematics, particularly statistics and linear algebra, will also be beneficial for understanding data analysis and machine learning concepts. Access to a computer with Python installed or a cloud-based coding platform like Google Colab is recommended to follow along with the practical exercises.

MODULE 1

FOUNDATIONS OF DATA SCIENCE AND PYTHON PROGRAMMING

We are about to kick start our journey in the realm of Data science, **where numbers tell stories, Python does the heavy lifting, and 'clean data' is basically a mythical creature.** In this module, we'll decode the secret language of data nerds (yes, 'Pandas' is a library, not a zoo animal), wield Python like a coding wizard, and learn why 'NaN' isn't just a bread brand but your worst spreadsheet nightmare. By the end, you'll be shouting *'I see patterns everywhere!'*, even in your coffee and soup stain.

This module serves as your gateway into the exciting world of data science. By the end, readers
will be able to:

1. Explain data science concepts.
2. Write basic Python code for data tasks.
3. Use Jupyter Notebook for analysis.
4. Differentiate Python from R/AI/ML.

1.1 What is Data Science?

Definition: *Data Science is an interdisciplinary field that combines statistics, mathematics, programming, and domain expertise to extract meaningful insights from structured and unstructured data. It involves techniques such as data cleaning, data visualization, machine learning (ML), and artificial intelligence (AI) to analyze patterns, make predictions, and drive decision-making.*

Data science is like a superhero team where a variety of skills; statistics, math, coding, and industry knowledge combine forces to uncover hidden stories in data, whether it's neatly organized or messy. The essence of data science lies in cleaning up chaotic data, transforming it into meaningful insights, and creating visual representations that make complex numbers more digestible. It also involves teaching computers to spot patterns through machine learning and using artificial intelligence to predict future trends, helping businesses and organizations make better decisions.

You can think of data science as possessing four superpowers: **Data wrangling**, which involves taming and cleaning wild, messy data; **Visual storytelling**, where numbers and data are made to reveal their true colors and insights; **Pattern recognition**, which finds hidden clues within the data to uncover valuable trends; and **Future-predicting**, which works like a crystal ball, except it's based on math and analytics, forecasting trends and informing decisions for the future.

Imagine you're a detective, but instead of solving crimes, you're solving business mysteries using data as your clues. Data science is this magical mix of number-crunching (statistics), problem-solving (math), coding skills (programming), and real-world know-how that helps us make sense of both neat spreadsheets and messy social media posts.

It's like being a data janitor (cleaning up messy information), a data artist (creating beautiful charts), and a fortune teller (using machine learning to predict trends) all rolled into one. Whether we're spotting customer patterns, forecasting sales, or helping doctors diagnose diseases, we're basically turning raw numbers into "aha!" moments that drive smarter decisions. The real magic

happens when we take all these pieces - cleaning, visualizing, analyzing, and predicting and turn them into actionable insights that can change how businesses and organizations operate.

At its core, data science is about asking better questions of our data and letting it tell us stories we might otherwise miss. And just like any good story, the better we understand the characters (our data points) and the plot (trends and patterns), the more meaningful our ending (business decisions) becomes.

The Key components of data science include:

- **Data Collection & Cleaning:** Gathering raw data from various sources and processing it
- **Exploratory Data Analysis (EDA):** Understanding data patterns through visualization and statistical methods.
- **Machine Learning & AI:** Developing models to predict outcomes and automate tasks.
- **Big Data & Cloud Computing:** Handling large datasets using advanced computing resources.

Data Science is widely applied in fields like healthcare, finance, cybersecurity, and business intelligence to improve efficiency and innovation.

1.2 Data Science VS Artificial Intelligence VS Machine Learning

Data Science, Artificial Intelligence (AI), and Machine Learning (ML) are sometimes confused. They are interconnected but distinct fields. **Data Science** focuses on extracting insights from data using statistics, visualization, and machine learning techniques. **AI** is a broader field that aims to create intelligent systems capable of reasoning, decision-making, and problem-solving. **ML**, a subset of AI, involves training algorithms to learn

patterns from data and make predictions without explicit programming. While Data Science leverages ML and AI for data-driven insights, AI encompasses various subfields beyond ML, such as robotics and expert systems.

Data Science is like the bartender who carefully measures each ingredient (data) to craft the perfect cocktail (insights). AI, on the other hand, is the overconfident regular who claims they can "mix anything," much like robots that almost don't fall over. Then there's Machine Learning (ML), the bar's creepy, loyal customer who memorizes everyone's orders after just a few visits, "You'll have the NaN-tini again, right?"

While they're all related, their roles are distinct: **Data Science** explains the why behind the mess (stats), answering, "Here's why you're drunk." **AI** is the ambitious dreamer, saying, "Let's build a robot to drink for you (and watch it fail hilariously)." **ML**, however, has learned from experience, predicting with, "I knew you'd order tequila... because you always do."

The moral of the story? AI dreams big, ML learns from past mistakes, and Data Science is the one who cleans up the mess. "No, the robot did NOT 'accidentally' order 100 margaritas."

1.3 Data Analysis vs Data Science vs Machine Learning

Data Science and Data Analysis are closely related but differ in scope and application. **Data Science** is a broad, interdisciplinary field that combines statistics, machine learning, and programming to extract insights, build predictive models, and develop AI-driven solutions. It deals with both structured and unstructured data, using advanced techniques like deep learning and big data

processing. **Data Analysis**, on the other hand, is a subset of data science focused on examining, cleaning, and visualizing data to uncover trends and support decision-making. While data analysts interpret historical data for insights, data scientists build models to predict future outcomes and automate processes.

They are almost the same because they share the same goal, which is to derive insights from data and use it for better decision making.

In fact, data science became popular as a result of the generation of chunks of data coming from online sources and activities (search engines, social media).

Being a data scientist sounds way cooler than being a data analyst. Although the job functions might be similar and overlapping, it all deals with discovering patterns and generating insights from data.

Data Science and Data Analysis are like cousins at a family reunion. While they're both related and share similar interests, they're often found in different corners of the room. Data Science is the cool, tech-savvy cousin who knows all about machine learning, AI, and deep learning, while Data Analysis is the practical one who focuses on rolling up their sleeves to clean up data and dig for those nuggets of insight. Data Science walks into a room and says, 'I'm going to predict the future with AI!' Meanwhile, Data Analysis sits quietly in the corner with a spreadsheet and says, 'I'm going to find trends in what's already happened.'"

"Both cousins are after the same thing: helping organizations make smarter decisions. But while Data Science is out there building

fancy models and speaking in code, Data Analysis is busy crunching numbers and making sure the data looks good on paper."

"Let's face it, saying 'I'm a Data Scientist' at a party does sound cooler than 'I'm a Data Analyst.' It's like saying you're a wizard versus an accountant, even though you both work with magic (data, that is). But at the end of the day, whether you're analyzing historical data or predicting future trends, it's all about uncovering insights and using data to make better decisions."

1.4 What is programming language?

A programming language in data science is like a magic wand that helps data scientists turn chaos into clarity. Python, R, SQL, and friends are the trusty tools that clean messy data, build machine learning models, and create shiny graphs,kind of like how a good coffee helps you make sense of your inbox. Python's the rockstar here, loved for being simple and having a ton of libraries, while others are like the quirky sidekicks, each with their own special powers. Together, they help turn raw data into decisions faster than you can say "predictive model!"

1.5 Python Language

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. With a vast ecosystem of libraries such as NumPy, Pandas, TensorFlow, and Scikit-learn, Python is widely used in web development, data science, artificial intelligence, machine learning, automation, and cybersecurity. Its

ease of learning and extensive community support make it one of the most popular programming languages today.

In today's digital age, Python is widely used in the field of data science. Its simplicity, extensive libraries, and active community make it an ideal choice for handling, analyzing, and visualizing data. It provides powerful libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn, which enable efficient data processing, statistical analysis, and machine learning implementation.

1.6 Python as an Interpreter

Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time.
Python is like that chill friend who loves to maintain its steeze and composure, doesn't like to rush. Instead of running off to do everything at once, it takes its time, executing one statement at a time. You don't need to worry about compiling anything, it's the laid-back version of coding. Just type python in the command line, and boom, you're in! It's like an interactive chat where Python listens to you, then carefully considers each sentence before responding. No pressure, no fuss, just pure, slow and steady code execution. Perfect for debugging... or if you like taking your sweet time with your code
The standard interactive Python interpreter can be invoked on the command line with the python command:

```
$ python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> sum = 7 + 9
>>> print(sum)
```

The >>> you see is the *prompt* where you'll type code expressions. To exit the Python interpreter and return to the command prompt, you can either type **exit()** or press Ctrl+Z then press ENTER While some Python programmers execute all of their Python code in this way using interpreter or command line style, those doing data analysis or scientific computing make use of IPython, an enhanced Python interpreter, or Jupyter notebooks, A web-based interactive environment for writing and running code, supporting python language originally created within the IPython project. We will be using jupyter notebooks for our lessons.

1.7 Python VS R

Python and R are like two superheroes in the data science world, each with their own special powers. Python is the versatile, all-rounder hero, loved for its simplicity, scalability, and vast libraries like Pandas and TensorFlow. It's perfect for machine learning, deep learning, and automating tasks basically, the go-to choice for large-scale, AI-driven projects. R, on the other hand, is the expert statistician, designed for statistical computing and data visualization. With powerful libraries like ggplot2 and dplyr, R shines in exploratory data analysis and statistical modeling, making it the favorite in academia and research. While Python dominates the industry, R is still the champion for in-depth, number-crunching analysis.

Learning Python or R (or any other programming language) requires several weeks and months. It's a huge time investment and you don't want to make a mistake. To get this out of the way, just start with Python because the general skills and concepts are easily transferable to

other languages. Well, in some cases you might have to adopt an entirely new way of thinking. But in general, knowing how to use Python in data science will bring you a long way towards solving many interesting problems.

1.8 Why Choose Python for Data Science & Machine Learning

Python is said to be a simple, clear and intuitive programming language. That's why many engineers and scientists choose Python for many scientific and numeric applications. Perhaps they prefer getting into the core task quickly (e.g. finding out the effect or correlation of a variable with an output) instead of spending hundreds of hours learning the nuances of a "complex" programming language. This allows scientists, engineers, researchers and analysts to get into the project more quickly, thereby gaining valuable insights in the least amount of time and resources. It doesn't mean though that Python is perfect and the ideal programming language on where to do data analysis and machine learning. Other languages such as R may have advantages and features Python has not. But still, Python is a good starting point and you may get a better understanding of data analysis if you use it for your study and future projects.

You are free to view Python as the comfy sweatpants of programming languages: easy to put on, fits well, and gets you to the important stuff fast. Scientists, engineers, and analysts love it because it's intuitive and lets them dive right into the core task, like figuring out how a variable affects an outcome, without getting bogged down in confusing syntax. You can go from zero to insights in no time, saving both time and brainpower. Sure, other languages like R have their own perks, but Python's ease of use makes it the perfect starting point for any data science project.

It's like choosing a smooth ride to get you to your destination faster, even if you know there are other fancy cars on the road.

1.9 Widespread Use of Python in Data Analysis

Python has become the dominant programming language for data analysis due to its simplicity, versatility, and an extensive ecosystem of libraries. Unlike traditional statistical tools such as R or MATLAB, Python offers an intuitive syntax that is easy to learn, making it accessible to both beginners and experienced data scientists. Its open-source nature ensures continuous improvements and contributions from a vast community of developers worldwide. Libraries such as NumPy and pandas enable efficient data manipulation and transformation, while Matplotlib and Seaborn provide powerful visualization tools. Furthermore, Python seamlessly integrates with databases, big data frameworks like Apache Spark, and cloud computing platforms, making it a highly scalable solution for handling large datasets. This flexibility has led to Python's adoption across various industries, including finance, healthcare, marketing, and cybersecurity, where data-driven decision-making is critical.

Another reason for Python's widespread use in data analysis is its strong support for machine learning and artificial intelligence (AI). With libraries such as Scikit-learn, TensorFlow, and PyTorch, Python allows data analysts and researchers to build predictive models with ease. The language also supports automation and scripting, reducing the manual effort required for repetitive data processing tasks. Additionally, its interoperability with other programming languages, such as C++ and Java, enables seamless integration into existing enterprise systems. Python's ability to handle structured and unstructured data, along with its support

for natural language processing (NLP) and deep learning, makes it indispensable for modern data analytics. As organizations increasingly rely on data to drive business strategies, Python's role in data analysis continues to grow, solidifying its position as the preferred language for data science professionals. Also, university graduates can quickly get into data science because many universities now teach introductory computer science using Python as the main programming language. The shift from computer programming and software development can occur quickly because many people already have the right foundations to start learning and applying programming to real world data challenges.

Python didn't just win the data science crown, it bribed the judges with its "readable syntax" and an army of libraries that do everything except make coffee (though there's probably a PyBrew module for that). It's the only language where you can go from writing "Hello World" to training a neural network in the time it takes MATLAB to compile "Good morning." With pandas for data wrestling, Matplotlib for "artistic" bar charts, and scikit-learn for when you want to predict stock markets but actually just predict which coworker will steal your lunch, Python is basically the duct tape of the digital world, holding together everything from quick scripts to AI that may or may not take over the world. The best part? When your code fails (and it will), the error messages are slightly less terrifying than Java's. Now if only it could fix your Imposter Syndrome too...

1.10 Is Mathematical Expertise Necessary for Data Science?

Data science often involves working with numbers and extracting valuable insights, but do you really need to be a math genius?

Well, sort of... but not *that* sort of. Here's the deal: while mathematical expertise is certainly important, the level of math knowledge you need really depends on what you're doing. If you're just analyzing data, performing basic analysis, or using pre-built machine learning models, you don't need to be able to recite every formula from memory (unless you're trying to impress at a data science party, but even then, who's got the time?). Think of it like cooking: you don't need to invent new recipes, but knowing how to balance flavors helps you create something delicious. Similarly, knowing linear algebra, statistics, and probability basics is helpful, but you don't need to be able to explain a Fourier transform to your grandma.

Now, if you're diving into AI, deep learning, or building new algorithms, you might want a stronger grasp of calculus and advanced math. This is where things get a little more like being a mad scientist in a lab, here, the math helps you create the magic. But hey, even if you can't solve for X in your sleep, don't panic! Most data scientists aren't holed up in rooms with chalkboards filled with equations. Thanks to tools and libraries like Python's NumPy or Scikit-learn, the heavy math lifting is done for you, leaving you free to focus on the fun stuff, getting insights from the data without needing to be the next Einstein.

It's not about being a math wizard, but about understanding just enough to make it work. As the saying goes: "You don't need to know how to build a rocket to ride in one, but a little math helps you not accidentally end up in space." You can definitely survive with a basic understanding of linear algebra, but as long as you can Google the math behind a machine learning model and know when your data's going off the rails, you're good to go. After all, math in data science is like the seasoning in a dish: you need just

enough to make it work, but too much, and it's overcooked. So, don't worry if you didn't ace your calculus exam. Just make sure you can explain why your model predicted your cat would run for president.

Einstein may have revolutionized physics without TensorFlow, but we're pretty sure he'd still need help debugging a Python script. The bottom line? You don't need to be the next mathematical prodigy. Just focus on the key concepts and let the libraries do the heavy lifting.

1.11 Integrated Development Environments (IDE)

For any programmer, and by extension, for any data scientist, the integrated development environment (IDE) is an essential tool. IDEs are designed to maximize programmer productivity. Thus, over the years this software has evolved in order to make the coding task less complicated. Choosing the right IDE for each person is crucial and, unfortunately, there is no "one-size-fits-all" programming environment. The best solution is to try the most popular IDEs among the community and keep whichever fits better in each case. In general, the basic pieces of any IDE are three: the editor, the compiler, (or interpreter) and the debugger. Some IDEs can be used in multiple programming languages, provided by language-specific plugins, such as Netbeans or Eclipse. Others are only specific for one language or even a specific programming task in the case of Python, there are a large number of specific IDEs, both commercial (PyCharm, WingIDE ...) and open-source. The open-source community helps IDEs to spring up, thus anyone can customize their own environment and share it with the rest of the community. For example, Spyder (Scientific

Python Development Environment) is an IDE customized with the task of the data scientist in mind.

1.12 Web Integrated Development Environment (WIDE): Jupyter Notebook

A **Web Integrated Development Environment (WIDE)** is an online or browser-based platform that allows users to write, execute, and manage code without the need for a traditional desktop IDE. One of the most popular WIDEs for data science and machine learning is **Jupyter Notebook**, which provides an interactive computing environment that supports multiple programming languages.

Jupyter Notebook is an open-source, web-based tool that enables users to write and execute code, visualize data, and document their workflow in a single interface. Unlike traditional IDEs that run locally, Jupyter operates within a web browser, making it accessible from any device with an internet connection. This feature makes it a preferred choice for collaborative projects, research, and online learning.

One of the key advantages of Jupyter Notebook as a WIDE is its **interactive execution.** Users can write and execute code in small blocks called **cells,** allowing for step-by-step debugging and experimentation. It also supports **rich output formats**, including data visualizations, markdown documentation, LaTeX equations, and interactive widgets, making it ideal for data analysis and machine learning projects.

Additionally, Jupyter Notebook supports multiple programming languages through **kernels**, with Python being the most commonly used. It integrates seamlessly with cloud platforms like

Google Colab and **Binder**, allowing users to run their notebooks online without requiring local installation. This cloud compatibility enhances accessibility and makes it easy to share projects with others.

To start using Jupyter Notebook, users can install it via **pip install jupyter** and launch it with **jupyter notebook**, which opens an interface in a web browser where they can create and manage notebooks (.ipynb files) or download **ANACONDA SOFTWARE** which has Jupyter Notebook as one of its packages. Its ability to combine **code execution, documentation, and visualization in a single environment** makes Jupyter Notebook a powerful WIDE for data science, machine learning, and research.

1.13 Get Started with Python for Data Scientists

Throughout this book, we will come across many practical examples. In this module, we will see a very basic example to help get started with a data science ecosystem from scratch. To execute our examples, we will use Jupyter notebook, although any other console or IDE can be used.

Setting Up the Python Environment

Before diving into data science with Python, it is important to set up the working environment.

Installing Python

Python can be downloaded from the official Python website https://www.python.org. It is recommended to install the latest stable version

Using Anaconda for Data Science:

Anaconda is a popular distribution that simplifies package management and deployment. It comes with essential libraries pre-installed. In this book, we will be using Anaconda.

Installing Anaconda:

1. Download Anaconda from https://www.anaconda.com
2. Follow the installation steps for your operating system (Windows/Mac/Linux).
3. Open the Anaconda Navigator or use the command line interface.

Want to learn Python faster? Just pour more hours into it like a squirrel on a sugar high. But remember, thinking like a programmer takes time, and cheat sheets are your new BFFs when you get lost. Even pros don't know everything, Google's got their back. And don't stress about learning everything now! It's like a buffet/ Start small, then dive deeper when you want to wow your boss or crush that job interview.

1.14 The Jupyter Notebook Environment

One of the major components of the Jupyter project is the *notebook*, a type of interactive document for code, text (with or without markup), data visualizations, and other output. The Jupyter notebook interacts with *kernels*, which are implementations of the Jupyter interactive computing protocol in any number of programming languages. Python's Jupyter kernel uses the IPython system for its underlying behavior.

Running the Jupyter Notebook

Once installed, you can start Jupyter Notebook by opening a terminal (Command Prompt or Anaconda Prompt) and running:

```
(base) C:\Users\HP > jupyter notebook
```

The command (jupyter notebook) will:

- Start the Jupyter Notebook server.
- Automatically open a web browser with the Jupyter interface.
- Display a file explorer where you can create or open. ipynb notebook files.

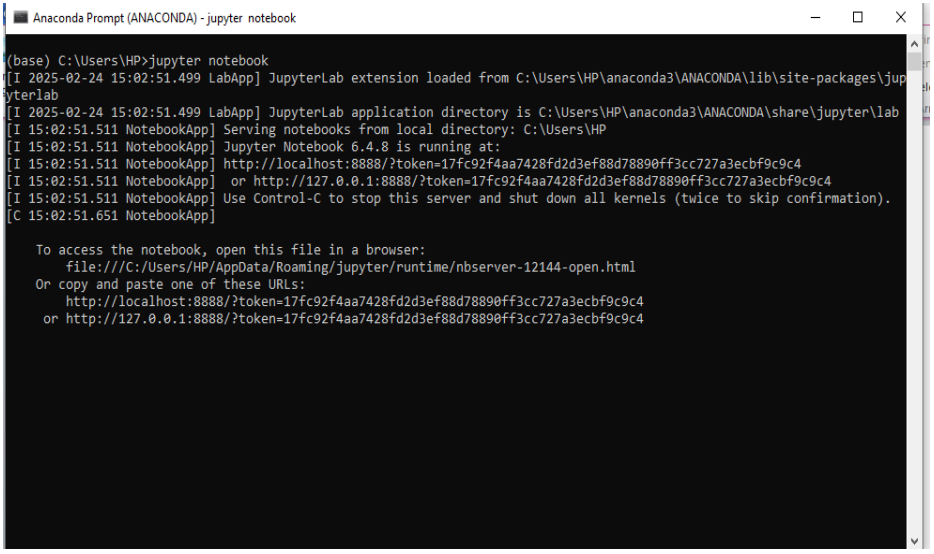

Anaconda Prompt (ANACONDA) - jupyter notebook
(base) C:\Users\HP>jupyter notebook
[I 2025-02-24 15:02:51.499 LabApp] JupyterLab extension loaded from C:\Users\HP\anaconda3\ANACONDA\lib\site-packages\jupyterlab
[I 2025-02-24 15:02:51.499 LabApp] JupyterLab application directory is C:\Users\HP\anaconda3\ANACONDA\share\jupyter\lab
[I 15:02:51.511 NotebookApp] Serving notebooks from local directory: C:\Users\HP
[I 15:02:51.511 NotebookApp] Jupyter Notebook 6.4.8 is running at:
[I 15:02:51.511 NotebookApp] http://localhost:8888/?token=17fc92f4aa7428fd2d3ef88d78890ff3cc727a3ecbf9c9c4
[I 15:02:51.511 NotebookApp] or http://127.0.0.1:8888/?token=17fc92f4aa7428fd2d3ef88d78890ff3cc727a3ecbf9c9c4
[I 15:02:51.511 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 15:02:51.651 NotebookApp]
To access the notebook, open this file in a browser:
file:///C:/Users/HP/AppData/Roaming/jupyter/runtime/nbserver-12144-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=17fc92f4aa7428fd2d3ef88d78890ff3cc727a3ecbf9c9c4
or http://127.0.0.1:8888/?token=17fc92f4aa7428fd2d3ef88d78890ff3cc727a3ecbf9c9c4

Figure 1.1: The Anaconda prompt

Creating a New Notebook

- In the Jupyter interface, click on "**New**" (top-right corner) and select "**Python 3**" (or another installed kernel).
- A new notebook will open, where you can write and execute code in cells.

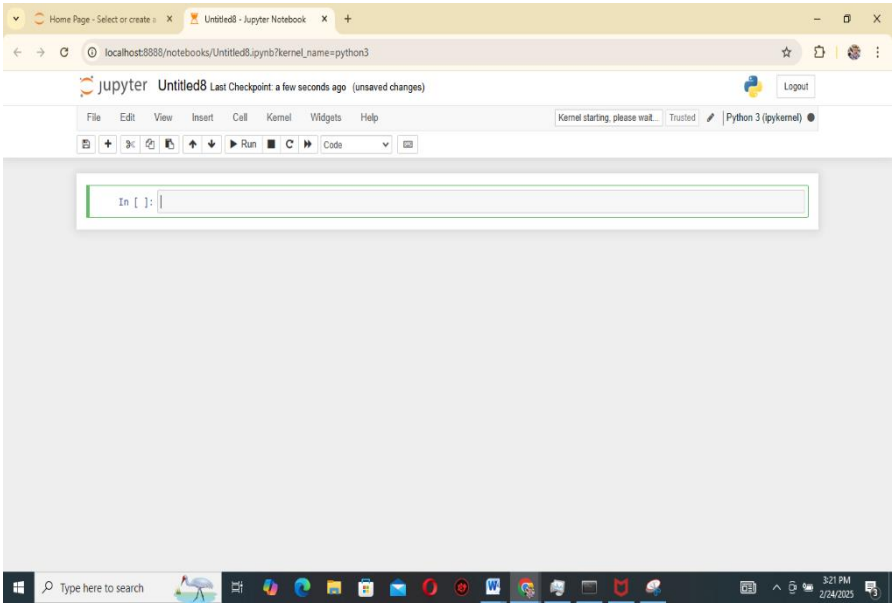
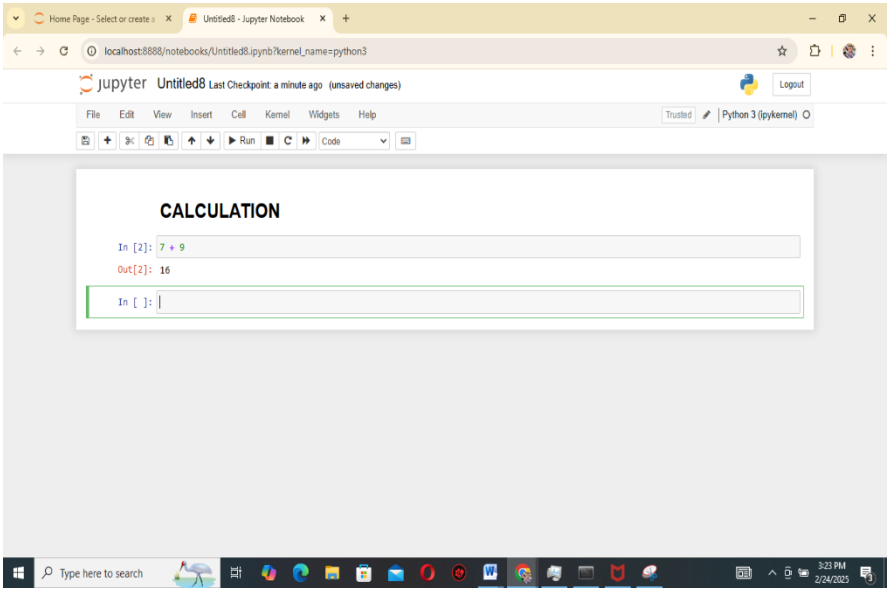


Figure 1.2: The Jupyter Notebook Environment

Running Code in Jupyter Notebook

- Type Python code in a **code cell** and press **Shift + Enter** to execute it.
- Use **Markdown cells** to write formatted text and document your code.

Figure 1.3: running my first code



Stopping Jupyter Notebook

- To stop Jupyter Notebook, click File > Close and Halt inside the notebook.
- In the terminal, press **Ctrl + C** and type **Y** to shut down the Jupyter server.

1.16 Writing our first python code: hello world

```
In [1]: Print ('Hello World")
Out [1]: Hello World
```

That's it! You've just made your Python debut! It's like learning to ride a bike, super simple, but it opens the door to all kinds of exciting adventures in programming. Just hit "run," and voilà! You've officially joined the Python party.

Renaming the filename:

First of all, we are going to change the name of the notebook 'Untitled 1' to something more appropriate 'MyFirstCode'. To do this, just click on the notebook name and rename it: '*MyFirstCode*' Let us begin by importing those toolboxes that we will need for our program. In the first cell we put the code to import the *Pandas* library as pd. This is for convenience; every time we need to use some functionality from the Pandas library, we will write pd instead of pandas. We will also import the two core libraries mentioned above: the numpy library as np and the matplotlib library as plt.

```
In [1]: import pandas as pd
            import numpy as np
            import matplotlib as plt
```

At this stage, we haven't performed any operation other than just importing libraries.

We may want to know our current working directory in order to access files easily.do this

```
In [1]: import os
        os.getcwd()
Out [1]: 'C:\\Users\\HP'
```

1.17 Tab Completion

This is a feature that helps users quickly complete variable names, function names, and object attributes while coding. It enhances productivity by reducing typing errors and improving code discovery.

How Tab Completion Works:

Variable & Function Name Completion

If you've already defined a variable or function, typing part of its name and pressing Tab will suggest possible completions.

Example:

```
In [1]: my_variable = 10
        my_var  # Press Tab here
```

This will complete it to my_variable.

Method and Attribute Completion

You can use Tab after a dot (.) to see available methods and attributes of an object.

Example:

```
In [1]: my_list = [1, 2, 3]
        my_list.  # Press Tab here
```

This will show a dropdown list of available methods like append(), clear(), copy(), *etc.*

Module & Package Completion

When importing modules, pressing Tab helps list available submodules.

Example:

```
In [1]: import numpy as np
        np.  # Press Tab here
```

This will display functions like array, mean, sum, *etc.*

File Path Completion (in functions like open())

If you're dealing with file paths, Tab can complete directory and file names.

Example:

```
In [1]: open("data/  # Press Tab here
```

It will show available files inside the data *folder.*

1.18 Introspection

Introspection is the ability of Python (and Jupyter Notebook) to examine objects, their attributes, methods, and documentation at runtime. This helps in understanding how to use functions, classes, and modules without referring to external documentation.

Introspection in Python (and Jupyter Notebook) is like having a super nosy friend who can't stop asking "What does this do?" but in a good way. It lets you peek under the hood of objects, check out their attributes, methods, and even read their user manual (no Googling required). So, if you're ever unsure how something works, just ask Python,it's got the answer, no external documentation required! It's like being able to talk to your code without needing a degree in cryptography.

Key Introspection Techniques in Jupyter Notebook

Using ? for Documentation

1. Appending ? to a function, variable, or module name displays its docstring.

Example:

```
In [1]: list?
```

This will show details about the list *class, including its methods.*

2. **For a specific method:**

```
In [1]: str.upper?
```

This displays the documentation for str.upper().

Using ?? for Source Code

?? not only shows the docstring but also the source code (if available).

Example:

```
In [1]: def sample_function():
            """This is a sample function."""
```

```
    return "Hello"
In [2]: sample_function??
```

This will display the function's definition along with its docstring.

Using** dir() **to List Attributes and Methods

dir() returns all attributes and methods of an object.

Example:

```
In [1]: dir(list)
```

This will list all methods like append, extend, pop, *etc.*

Using** type() **to Get the Object Type

Example:

```
In [1]: type(10)  # Output: <class 'int'>
In [2]: type("Hello")  # Output: <class 'str'>
```

Using help() for Detailed Documentation

help() provides detailed documentation, similar to ?, but in an interactive format.

Example:

```
In [1]: help(dict)
```

1.19 Import Conventions

In Python, importing modules follows standard conventions to improve code readability and maintainability.

1. Standard Import Convention

- Modules are typically imported at the beginning of a script.

Example:

```
In [1]: import os
    import sys
```

2. Importing with Aliases

Importing with aliases is like giving your favorite tools nicknames so you don't have to say the full name every time. For example, instead of typing matplotlib.pyplot, you can just call it plt. It's quicker, saves you from typing a lot, and makes you sound like

you know what you're doing. Plus, the Python community has adopted these shorthand conventions, so when you see pd or sns, you'll know exactly what's going on, like a secret handshake among programmers!

The Python community has adopted a number of naming conventions for commonly used modules in data science.

Example:

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

When you see np.arange, this is a reference to the arange function in NumPy. This is done because it's considered bad practice in Python software development to import everything (from numpy import *) from a large package like NumPy.

3. Importing Specific Functions

Only necessary functions are imported to avoid unnecessary memory usage. Importing specific functions in Python is like only grabbing the ingredients you need for your recipe,why stockpile a whole pantry if you just need salt and pepper? By selectively importing only the functions you need, you save memory and keep your code nice and tidy, avoiding the clutter of unused tools that you'll never touch. It's efficient, and your program will thank you later!

Example:

```
In [1]: from math import sqrt, pi
```

4. Importing All (*) – Not Recommended

Avoid using from module import * as it can lead to conflicts.

Using from module import * in Python is like inviting everyone to the party without checking who's coming. Sure, it seems convenient, but suddenly you've got name conflicts, messy code,

and people bumping into each other (metaphorically speaking). It's better to be specific about who you're inviting to the function, method, or class party. This way, you avoid chaos and keep things organized.

Example (not recommended):

```
In [1]: from math import *  # Pollutes namespace
```

5. Multi-line Imports for Readability

Multi-line imports are like packing for a trip you want to fit everything in without making it a mess. When you're importing several items, you can use parentheses to spread things out over multiple lines, making your code neat and easy to read. It's like giving your imports a bit of breathing room instead of cramming them all into one line like an overstuffed suitcase.

When importing multiple items, use parentheses.

Example:

```
In [1]: from collections import (Counter,
                                 OrderedDict,
                                 defaultdict)
```

Example

```
from math import (
    pi,
    sqrt,
    cos,
    tan
)
```

1.20 Language Semantics

The Python language design is distinguished by its emphasis on readability, simplicity, and explicitness. Some people go so far as to liken it to "executable pseudocode."

Indentation, not braces

Indentation in Python refers to the spaces or tabs used at the beginning of a line to define the structure of the code. Unlike other programming languages that use curly braces {} or keywords to indicate code blocks, Python relies on indentation to determine the grouping of statements. Proper indentation is essential because Python does not allow incorrectly indented code; otherwise, it raises an **IndentationError**. The standard convention in Python is to use **four spaces per indentation level**, although tabs can also be used (but mixing tabs and spaces should be avoided). Indentation is primarily used in control structures like loops (for, while), conditionals (if, elif, else), functions (def), and class definitions (class).

```
In [1]: def greet(name):
            if name = = ”Dr Clive”:
          print("Hello,", name)
            else:
          print(name, "You’re not Clive")
```

Python’s design focuses on readability and simplicity, with indentation used instead of braces to define code blocks. Indentation, typically four spaces, is crucial to Python’s structure, mess it up, and you’ll get an *IndentationError*. It’s used in loops, conditionals, functions, and class definitions to keep your code neat and organized, like a well-arranged desk (but without the clutter).

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block.

In this example, indentation determines the structure of the function and the conditional statements. Without proper indentation, the code would not run. Since Python enforces

indentation as part of its syntax, it improves code readability and maintainability, making it easy to understand the flow of execution.

As you can see by now, Python statements also do not need to be terminated by semi- colons. Semicolons can be used, however, to separate multiple statements on a single line:

```
In [1]: a = 5; b = 6; c = 7
```

Putting multiple statements on one line is generally discouraged in Python as it often makes code less readable.

1.21 Everything is an object

In Python, everything is an object yes, even numbers, strings, lists, and that one function you wrote at 2 AM that you barely remember. It's like the "everything is a bagel" of programming. This object-oriented design makes data science a breeze. For example, NumPy arrays, Pandas DataFrames, and matplotlib plots are all objects, each with its own little personality and skills, making data manipulation feel like a magic trick. Thanks to Python's dynamic typing and object-oriented powers, you can wrangle large datasets, apply transformations, and build machine learning models as if you're assembling a data science Avengers team with libraries like Scikit-learn and TensorFlow. It's like having a toolbox where every tool knows exactly what to do, no need to wrestle with the wrench every time!

1.22 Comments

In Python, comments are like the helpful notes you leave for yourself or your future self when you forget what your code was doing (which happens a lot, trust me). They explain what's going on in the code, making it easier for others to understand and for you to avoid scratching your head in confusion a few weeks later.

Python has two types: single-line comments, which start with a # (kind of like writing “don’t forget the coffee” on your to-do list), and multi-line comments, which use triple quotes (''' or """) for when you have a lot to say, like a motivational speech to your code before running it. The best part? The Python interpreter totally ignores them,it's like talking to your plants, they won't judge you!

At times you may also want to exclude certain blocks of code without deleting them. An easy solution is to comment out the code:

```
In [1]: # This is a single-line comment
        x = 10  # Inline comment
        """
        This is a multi-line comment,
        often used for documentation.
        """
```

1.23 Function and object method calls

A function is like your personal robot that does the heavy lifting for you without complaining. You give it a task, it performs it, and then hands you the result,no questions asked. Functions are defined with the *def* keyword, followed by a name (because every robot needs a name, right?), and they can take parameters (the instructions you give to your robot). When it's time for your robot to work, you just call it by its name followed by parentheses, and boom,task completed. If the robot needs any special tools (i.e., arguments), you just pass them along! It’s the ultimate life hack to avoid repeating yourself.

```
In [1]: def greet(name):
            return f"Hello, {name}!"
In [2]: print(greet("Clive"))  # Function call
Out [2]: Hello Clive
```

An **object method call** refers to invoking a function that belongs to a specific object (an instance of a class). Methods are functions defined inside a class and operate on the object's attributes. They are called using the **dot notation (.)**.

Think of an object method call as asking your pet robot to perform a task, but this time, it's got a personalized twist. When you create an object (let's call it a "robot"), it comes with built-in functions,called methods,that help it do things like fetch your slippers or update your status on social media. These methods live inside the object, and to make them work, you just use the dot notation (a fancy way of saying "hey robot, do this for me"). So, if your object is *robot*, and it has a method called *clean*, you just say *robot.clean()*, and voilà,your robot starts cleaning! It's like telling a dog to fetch, except way less slobber.

Example of an object method call:

```
In [1]: class Person:
                        def __init__(self, name):
                        self.name = name

                        def greet(self):
                        return f"Hello, {self.name}!"

            p = Person("Clive")
In [2]: print(p.greet())  # Object method call
```

We will discuss more on this later in this book.

QUESTIONS

1. What is the main goal of data science?
2. Name two key differences between **data science** and **data analysis**.
3. Why is "clean data" often called a "mythical creature"?
4. What does **EDA** stand for, and why is it important?
5. How does machine learning differ from traditional programming?
6. Why is Python preferred over R for large-scale data tasks?
7. What does `NaN` mean in a dataset, and why is it problematic?
8. Name two Python libraries used for data manipulation.
9. What is the purpose of **Jupyter Notebook**?
10. Write a Python code snippet to print "Hello, Data World!".
11. What is the main advantage of using **Anaconda** for data science?
12. How do you launch a Jupyter Notebook from the command line?
13. What does `import pandas as pd` do?
14. Name one Python library for data visualization.
15. How is a **.ipynb** file different from a regular Python script?
16. Give one reason why Python is more beginner-friendly than R.
17. How is **AI** broader than **machine learning**?
18. What makes Python better for **big data** than Excel?
19. Name a task where **R** might outperform Python.
20. Python's comment system (`#` and docstrings) is often dismissed as trivial, but in data science workflows, improper commenting can create catastrophic misunderstandings. Analyze how each of these scenarios could lead to failures in a

collaborative data pipeline, and propose specific commenting strategies to prevent them

MODULE 2

GETTING STARTED WITH PYTHON: VARIABLES, DATA TYPES AND FUNCTIONS

This module dives into the magical world of Python, where you'll meet the essentials: variables, data types, and functions, plus a sprinkle of essential libraries (just the basics, don't worry). By the end of this adventure, you'll be able to:

1. **Understand and Use Variables in Python**
    - Define and assign values to variables.
    - Understand variable naming conventions and best practices.
    - Work with dynamic typing and type inference in Python.
2. **Comprehend Python Data Types**
    - Understand and differentiate between fundamental data types (integers, floats, strings, booleans, etc.).
    - Work with complex data types (lists, tuples, dictionaries, and sets).
    - Perform type conversions and casting.
3. **Master Basic String Operations**
    - Concatenate and format strings.
    - Utilize string methods for manipulation.
4. **Understand and Write Functions in Python**
    - Define functions using the def keyword.
    - Use function arguments and return values.
    - Work with default arguments, keyword arguments, and variable-length arguments.

- Understand the scope of variables (local vs. global).

VARIABLES

A variable in Python is like a box where you can stash your stuff,except this box doesn't require any moving or cleaning. It's a memory location that holds data. When you assign a variable (aka give it a name), you're essentially saying, "Hey, this box will hold this thing over here on the right side of the equals sign!" So, if you assign x = 5, Python's like, "Got it! The box named 'x' now holds the number 5, and I'll be using that box whenever you need it." It's like giving your cat a name tag... but instead of a cat, it's data! When assigning a variable (or *name*) in Python, you are creating a *reference* to the object on the righthand side of the equals sign.

```
# Example of variables
In [1]: x = 10         # Integer
        y = 3.14       # Float
        name = "Alice"  # String
        is_active = True  # Boolean
```

In practical terms, consider a list of integers:

```
In [2]: a = [1, 2, 3]
```

Suppose we assign variable 'a' to a new variable 'b':

```
In [3]: b = a
```

In some languages, this assignment would cause the data [1, 2, 3] to be copied. In Python, a and b actually now refer to the same object, the original list [1, 2, 3]

Variable naming convention

Variable naming conventions help make code more readable, maintainable, and error-free. View this practice as one giving your code a neat, organized closet instead of a pile of laundry. They

make your code more readable, maintainable, and (hopefully) less likely to give you a headache.

Key principles for naming variables include:

1. **Use Descriptive Names** – Name your variables something that actually tells you what they are. Instead of x, try number_of_apples or user_age. Your future self will thank you.
2. **Follow Case Styles** – Common styles include:
    - **Snake Case** (my_variable) – Used in Python.
    - **Camel Case** (myVariable) – Used in JavaScript and Java.
    - **Pascal Case** (MyVariable) – Used for class names.
3. **No spaces**: Python doesn't do spaces in variable names. If you want to separate words, use underscores (e.g., my_variable) instead of trying to sneak in a space like it's an exclusive party.
4. **Start with a Letter or Underscore** – Variable names should begin with a letter (A-Z, a-z) or an underscore (_) but not a number.
5. **Avoid Reserved Keywords** – Do not use Python keywords like class, def, or return as variable names.
6. **Use lowercase**: By convention, variable names are written in lowercase. No shouting with uppercase to the interpreter (unless you're feeling super excited, but it's not appropriate for variables).
7. **Maintain Consistency** – Stick to a consistent naming style across your project.

Following these conventions is like organizing your desk,suddenly, everything has a place, and you can actually find what you need without digging through a pile of messy papers. It not only makes your code look more professional but also cuts

down on bugs and confusion. It's like giving your future self a roadmap so you don't wander into the land of "Why doesn't this work?" every time you revisit your code. Plus, your colleagues (or your future self) will love you for it when they can read your code without needing a secret decoder ring.

DATA TYPES

Variables are like your kitchen pantry, and data types are the ingredients you store inside them. Now, what exactly is a data type? It's essentially the kind of ingredient you're working with: Is it a number you can bake into a cake, a word you can toss into a soup, or maybe a list of ingredients to cook up later?

In Python, data types are the classification of the value that a variable holds. These data types determine what kind of operations you can perform on them without causing a kitchen disaster (like trying to bake a list of ingredients instead of using it to store them).

Python supports several built-in data types, each as useful as a different ingredient in your coding recipe.

1. Integer (int): Whole numbers (e.g., 10, -5).
2. Float (float): Decimal numbers (e.g., 3.14, -0.001).
3. String (str): Text data (e.g., "Hello, World!").
4. Sequence data type (String, List Tuple, Range)
5. Mapping data type
6. Set data type
7. Boolean (bool): Represents True or False.
8. Binary
9. None

Note: Integer (int) and Float (float) are termed Numeric data type

It's crucial to know what kind of "ingredient" you're working with when cooking up your Python code! Knowing the data type of a variable helps you decide what operations can be safely

performed on it, without causing any confusion in the kitchen (or your code).

If you ever wonder what type of data a variable is holding, you can easily check it using Python's built-in type() function. Think of it as your "ingredient label" that tells you whether you're working with a string of text, a number, or something else entirely.

Here's how you can check the type:

```
# checking data types
print(type(x))         # Output: <class 'int'>
print(type(y))         # Output: <class 'float'>
print(type(name))      # Output: <class 'str'>
print(type(is_active)) # Output: <class 'bool'>
```

1. Numeric Data Type

In Python, numbers come in two main flavors: **int** and **float**. Think of **int** as the no-nonsense, reliable number that never worries about decimals, while **float** is the fancy, decimal-loving number who enjoys a little extra precision.

An **int** (short for integer) is perfect for whole numbers, and Python's got your back when it comes to handling *really* big numbers. Seriously, Python doesn't flinch when it encounters a number the size of your student loan debt (or any large number, really).

Here's how it works:

```
my_int = 1000000000000000000  # Python's idea of "a little number"
print(type(my_int))  # Output: <class 'int'>

my_float = 3.14159  # Because Pi is always there when you need it
print(type(my_float))  # Output: <class 'float'>
```

Example:

```
In [1]: ival = 10
In [2]: ival ** 2
Out[2]: 100
```

Floating-point numbers in Python are the drama queens of the number world. They come with decimal points, always demanding a bit more attention, and when things get *really* serious, they break out the scientific notation.

Python's **float** type stores floating-point numbers as double-precision (64-bit) values. This basically means they're precise and can handle a lot of decimals, but don't expect them to always be perfectly exact,floating-point numbers sometimes have a way of rounding up or down, just like your hopes for a pizza to show up in 30 minutes.

```
# Regular float
pi = 3.14159
print(type(pi))  # Output: <class 'float'>

# Scientific notation
big_number = 1.23e6  # 1.23 * 10^6, a fancy way of writing 1,230,000
print(big_number)  # Output: 1230000.0
```

In scientific notation, **1.23e6** is just Python's way of saying, "Yo, let's move that decimal point six places to the right!" It's like a shorthand for the big numbers you'll encounter, but without needing to write out all the zeros like a coding fanatic with too much time on their hands.

```
In [3]: fval = 7.243
In [4]: fval2 = 6.78e-5
```

In Python, when you perform integer division (using the // operator) but the result isn't a nice, neat whole number, Python

doesn't throw a tantrum,it just gracefully gives you a floating-point number instead. It's like trying to divide a pizza into uneven slices and Python saying, "Well, here's your result, even if it's not a pretty whole number!"

Here's how it works:

```
# Integer division with a remainder (produces float)
result = 7 / 2
print(result)  # Output: 3.5

# Integer division with the '//' operator
result2 = 7 // 2
print(result2)  # Output: 3

# If you want a float from integer division, Python's got you covered!
result3 = float(7 // 2)
print(result3)  # Output: 3.0
```

Notice that `7 / 2` produces a **float** (3.5), while `7 // 2` produces a **whole number** (3). But don't worry, if you really want a floating-point result, you can easily convert it using `float()`. Python's all about keeping you happy and your numbers neat, even when they get a little messy.

Ah, yes! If you're in the mood for some "old-school" division,like the C programming language style,Python's got you covered with the floor division operator //.

This operator ensures that any division that doesn't result in a whole number is neatly "floored," meaning it drops the decimal part and returns just the integer part.

It's like the difference between taking the "easy way" (normal division) and "getting down to business" (floor division).

For example:

```
# Floor division
result = 3 // 2
print(result)  # Output: 1
```

```
# But if you use regular division...
result2 = 3 / 2
print(result2)  # Output: 1.5
```

So, `3 // 2` gives you **1** (because it just drops the decimal part like a boss), whereas `3 / 2` gives you **1.5** (the regular float result).
Floor division is your go-to if you want to keep things **strictly integer**,no room for those pesky decimals

2. Strings

A **string** in Python is a sequence of characters enclosed in either single quotes (') or double quotes ("). It can contain letters, numbers, symbols, or even spaces. Think of a string as a fancy container that holds text, like a shopping bag full of words..
Ah, the beauty of strings in Python,where you can go all "quote-ception" and still get the job done! Whether you're a "single-quote fan" or a "double-quote enthusiast," Python lets you express your string literals using either. It's like picking your favorite flavor of ice cream,whatever makes you happy!

```
a = 'Using single quote in writing a string'
b = "This string uses double quote"
```

Both strings will work equally well, and Python won't judge you for your choice of quotes. It's like a chill coffee shop where you can order your drink however you want and still get the same great taste! The only catch? You should probably avoid mixing them up unless you're trying to start a fight with your code:

```
# This is fine:
string3 = "I said 'hello'!"
string4 = 'I said "hello"!'
# But this could cause problems:
string5 = 'I said "hello'  # Oops! We forgot the closing quote!
```

So go ahead,choose your quotes wisely, and let your strings flow!
For multiline strings with line breaks, you can use triple quotes, either ''' or """:

```
c = """
This is a longer string that
spans multiple lines
"""
```

You might be in for a little surprise,our friendly string c is sneakier than it looks! Although it just seems like a regular multi-line message, it's actually hiding four whole lines inside. Why? Because even the invisible line breaks after the opening triple quotes (""") and at the end of each line sneak their way into the string like uninvited party guests.

Wanna catch them in the act? Just call in the count() method like a detective on a mission:

```
In [7]: c.count('\n')
Out[7]: 3
```

Boom! It finds **three** newline characters skulking around in there. The fourth line? That's just the part after the last newline, quietly minding its business. Strings: they're not just text, they're little drama queens with hidden secrets

Python strings are immutable; In the world of Python, strings are divas,they do *not* like being changed directly! Once you've created a string, it's set in stone. Try to sneak in and change one of its characters like this:

```
In [8]: a = 'this is a string'
In [9]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
---> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment
```

Boom! Python throws a **TypeError** faster than you can say "oops!" Why? Because strings are *immutable*, which is a fancy way of saying: "Don't touch me,I'm perfect as I am!"

So, if you want to make changes, you've gotta play it smooth: create a brand new string using methods like replace(). Python strings are like royalty,you can't just walk up and change their wardrobe. You need to craft a whole new version of them!

```
In [10]: b = a.replace('string', 'longer string')
In [11]: b
Out[11]: 'this is a longer string'
```

Even after all that fancy replace() business, the original string a is still chilling with the big boys just the way you left it:

```
In [12]: a
Out[12]: 'this is a string'
```

Boom,'this is a string' pops right back out, totally unbothered. Strings in Python are like classy old-school vinyl records,you can remix them all you want, but the original track stays untouched.

Now, speaking of transformations, Python's got a magical tool called str() that can turn almost *anything* into a string. Got a float like 5.6? Toss it into str() like this:

```
In [13]: a = 5.6
In [14]: s = str(a)
In [15]: print(s)
Out [15]: 5.6
```

And voilà! 5.6 becomes a lovely string version of itself, all prim and proper for display. Python's got style *and* flexibility!

Strings in Python are basically sequences of Unicode characters,fancy talk for "they behave a lot like lists or tuples." You can break them apart, loop through them, or even slice them like a fresh loaf of bread.

For example:

```
In [16]: s = 'python'
In [17]: list(s)
Out[17]: ['p', 'y', 't', 'h', 'o', 'n']
```

This turns your sleek little string into a list: ['p', 'y', 't', 'h', 'o', 'n']. And if you're only in the mood for a taste of it, slicing steps in:

```
In [18]: s[:3]
Out[18]: 'pyt'
```

And bam,just the first three letters: 'pyt'. This slicing trick is pure magic and works across many Python sequences. It's like your string has a built-in deli counter!

Now, let's talk about the mysterious backslash \. It's not just a weird slanted line,it's an *escape artist*! Use it to sneak special characters into your strings like \n (newline) or \u2764 (a Unicode heart). But if you just want your backslashes to act like... well, **backslashes**, you'll need to double them up:

```
In [19]: s = '12\\34'
In [20]: print(s)
12\34
```

Output? 12\34. One backslash to escape the other. Yep, it's the buddy system.

Feeling overwhelmed by all those double slashes? Don't worry,Python's got your back with **raw strings**! Just add an r before your string and Python will stop trying to be clever:

If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately, you can preface the leading quote of the string with r, which means that the characters should be interpreted as is:

```
In [21]: s = r'this\has\no\special\characters'
In [22]: s
Out[22]: 'this\\has\\no\\special\\characters'
```

Now Python won't try to play detective and interpret those slashes,it just leaves them alone. What you type is what you get. The r stands for *raw*, but we like to think of it as *relax*, because it saves you from backslash madness.

Now let's talk about gluing strings together. If you have two separate strings and you want to make one mega-string, just add them like this:

```
In [23]: a = 'this is the first half '
In [24]: b = 'and this is the second half'
In [25]: a + b
Out[25]: 'this is the first half and this is the second half'
```

But wait, there's more! Ever wanted to sneak values into a string, like prices, quantities, or dramatic punchlines? That's where **string formatting** comes in. Python has a superhero method called .format() that lets you plug values into placeholders inside a string template.

```
In [26]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

In this string,

- {0:.2f} means to format the first argument as a floating-point number with two decimal places.
- {1:s} means to format the second argument as a string.
- {2:d} means to format the third argument as an exact integer.

To substitute arguments for these format parameters, we pass a sequence of arguments to the format method:

```
In [27]: template.format(4.5560, 'Argentine Pesos', 1)
Out[27]: '4.56 Argentine Pesos are worth US$1'
```

String formatting in Python is like giving your data a glow up before sending it out to a fancy party You can dress up your numbers give your words a trim and even align everything like it is standing in a military parade.

Want a number to show just two decimal places Python says Sure thing boss Want your text centered like it is meditating in yoga class Python nods and goes Namaste.

There are so many ways to format strings it is like having a closet full of outfits for your variables From old school percent style to

the stylish format method to the runway ready f strings Python has got it all

If you are ready to become a formatting fashionista and give your strings the red carpet treatment the official Python docs are like your personal stylist Just don't blame us when you start formatting everything including your grocery list.

3. **Set Data Types** (Unordered collections of unique elements)

Imagine a basket where you can throw in fruits but the basket is super picky It refuses duplicates You drop in an apple another apple and a banana and it just goes Nope I already have an apple So in the end your basket only holds one apple and one banana

That is basically how a set works

A set is an unordered collection of unique elements

That means

- No duplicates allowed
- The items have no specific order
- You can add or remove items but every element must be unique

There are two types of set vollections:

i. **Set** (set) –

This is your typical, laid-back collection of unique elements. It's like a VIP list where everyone gets in, but no one is allowed to repeat! So, if you try to add {1, 2, 3, 3}, Python will just keep {1, 2, 3} and ignore that second 3. Duplicates? Not in this club! Sets are **mutable**, meaning you can add or remove elements as you like.Collection of unique elements (e.g., {1, 2, 3, 3} → {1, 2, 3})

ii. **Frozen Set** (frozenset) – If a regular set is the free-spirited friend who loves to change things up, the **frozenset** is the

chill, unshakeable one. A frozenset is an **immutable** version of a set, meaning once you create it, you can't add, remove, or change any of its elements. It's like a set that's been frozen in time,perfect when you need to ensure that the set stays unchanged throughout your program.

Example:

```
unique_numbers = {1, 2, 3, 3}  # Set (duplicates removed)
immutable_set = frozenset([1, 2, 3, 4])  # Frozen set
```

4. **Sequence Data Types** (Ordered collections of data)

Sequence data types in Python are like well-organized files in a filing cabinet,everything has its place, and the order matters. There are four main types, each with its own special characteristics.

- **String (str)** – A string is a sequence of characters, essentially text. It's like a paragraph, a word, or even a single character, all wrapped up in quotes. You can use either single quotes ' or double quotes " to define a string.
- **List (list)** –A list is an ordered collection of items. It's like a to-do list or a shopping list,things can be added, removed, or changed freely. Lists are **mutable**, meaning you can modify them at will. They can hold different types of data, including numbers, strings, and even other lists. (e.g., [1, 2, 3, "text"])
- **Tuple (tuple)** –A tuple is very similar to a list, but with one crucial difference,it's **immutable**, meaning once it's created, you can't change, add, or remove elements. You can think of a tuple as a set of data that's locked in place, like a receipt that you can't edit after you've received it. (e.g., (1, 2, 3, "data"))
- **Range (range)** – A range represents a sequence of numbers, often used in loops. It's a very efficient way of

generating a series of numbers without creating a full list. You can specify the start, stop, and step of the range to generate numbers over a specific interval. (e.g., range(1, 10))

Example:

```
text = "Data Science"  # String
numbers = [1, 2, 3, 4]  # List
coordinates = (10, 20, 30)  # Tuple
seq = range(1, 5)  # Range
```

5. **Mapping Data Type** (Key-value pairs)

Imagine you're in a world where everything is organized by secret agents (keys) who each have a special item (value) to guard. The **Mapping Data Type** in Python is exactly like this: a collection of **key-value pairs** where each key (agent) holds its precious value (item). The most famous agent in this world is the **Dictionary (dict)**.

A **dictionary** is like a secret agent's file cabinet where each agent (key) has a specific file (value) assigned to them. The best part? No two agents can have the same file. Every key is unique, and each key is assigned exactly one value.

- **Keys**: Think of them as agents with ID cards,they must be unique and immutable (no changing agents in the middle of the mission)!
- **Values**: These are like the top-secret documents each agent is guarding. They can be anything from numbers to strings, lists, or even other secret agents (dictionaries!).

Example:

```
Agent_Delta = {"name": "Clive", "age": 25, "city": "Sapele"}
Print(Agent_Delta)
```

Accessing the Value – You just call the agent's ID (key), and bam! You get their file (value).

```
print(Agent_Delta["name"])  # Output: Clive
```

Adding or Updating Agents (Keys) – New agents can join the force, or existing ones can get their files updated. Maybe Clive gets a promotion to age 31!

```
Agent_Delta["country"] = "Nigeria"  # New agent on the team!
Agent_Delta["age"] = 31  # Update agent Clive's age
```

Removing Agents – If an agent has retired or gone rogue, you can delete them from the files.

```
del Agent_Delta["city"]  # Goodbye, agent "city"!
```

Check if an Agent Exists – You can also check if an agent exists before trying to access their file.

```
print("name" in Agent_Delta)  # Output: True (Agent "name" is here!)
```

6. **Boolean Data Type** (Logical values)

In the wild world of Python, there are two legendary siblings: **True** and **False**,the dynamic duo of decision-making! They're like the superheroes of logic, swooping in whenever you need a yes-or-no answer.

Did you ask if $5 > 3$? **True** flexes its muscles.

Is a banana a computer? **False** facepalms.

But wait,there's more! These two love teaming up with their sidekicks: `and` (the strict bouncer who only lets in *both* truths) and `or` (the chill friend who's happy with *either*).

```
In [1]: True and True
Out[1]: True
In [2]: False or True
Out[2]: True
```

True and **False** aren't just concepts,they're full-fledged values that can be assigned to variables. It's like giving a name tag to your

True and **False** friends and letting them do some heavy lifting in your code.

Example:

```
is_raining = True
is_sunny = False
print(is_raining)  # Output: True
print(is_sunny)    # Output: False
```

Now **is_raining** is **True**, and **is_sunny** is **False**. These variables can now be used in logical expressions, just like the heroes they are!

Example

```
is_weekend = True
is_tired = False

if is_weekend and not is_tired:
   print("Time to have fun!")
else:
   print("Maybe a nap first.")
```

In this example, we used our Boolean variables to check if it's the weekend and if we're not tired. If both conditions are true, we get the green light for fun!

7. **Binary Data Types** (Used for handling binary data)

Welcome to the world of **Binary Data Types** in Python! These types are specially designed to handle binary data efficiently, making them the go-to tools when you're working with data in byte form. Let's dive into the three main types that help manage binary data.

- **Bytes (bytes)** – First up, we have **bytes**, the tough guy of the group. Imagine a **stubborn vault** that you can't modify once it's locked. It's the kind of guy who gets the job done but doesn't take kindly to any changes. Once you create a

bytes object, it's solid and unbreakable. Think of it as your data that's been written in stone!

- **Bytearray (bytearray)** – Next up, meet **bytearray**, the flexible, shape-shifting superhero of the crew! This bad boy is mutable, meaning you can change things on the fly. If bytes is the immovable wall, **bytearray** is the superhero who **dances through walls**. Need to adjust some data? This is your go-to guy!
- **Memoryview (memoryview)** – Finally, we have **memoryview**. This is the ninja of the group,silent, efficient, and extremely resourceful. Think of it as someone who **sneaks in and tweaks the data without even touching the original source**. With memoryview, you're not copying data around like a hoarder; instead, you're accessing it **directly** with laser precision, saving memory like a pro.

Example:

```
b = bytes([65, 66, 67])  # b'ABC'
ba = bytearray([65, 66, 67])  # Mutable bytes
mv = memoryview(b)  # Memory-efficient view of bytes
```

8. **None Type** (Represents the absence of a value)

This is Python's way of saying, *"Yeah, there's literally nothing here."*

It's not zero.

It's not empty.

It's not false.

It's just... **None**. Like that one chair at the party nobody sits on, but it's still *technically* there.

What is none?

None is Python's version of **null**. It represents the **absence of a value**, like an invisible placeholder that politely says, "Nothing to see here, folks!"

```
In [9]: a = None
In [10]: a is None
Out[10]: True
In [11]: b = 5
In [12]: b is not None
Out[12]: True
```

If you create a function and forget to return something,don't worry! Python's got you covered. It'll quietly slip in a None for you like a waiter who clears your plate without asking.

```
def mysterious_function():
    pass

result = mysterious_function()
print(result)  # Output: None
```

See? No return? No problem. Python just goes, "Eh, let's make it None."

None is also a common default value for function arguments:

```
def add_and_maybe_multiply(a, b, c=None):
result = a + b
if c is not None:
result = result * c
return result
```

Note: None is not only a reserved keyword but also a unique instance of NoneType:

```
In [13]: type(None)
Out[13]: NoneType
```

So, the next time you encounter None, don't be alarmed. It's just Python's elegant way of saying,

"I didn't forget anything,I'm intentionally leaving this blank."

DYNAMIC REFERENCES, STRONG TYPES

```
In [12]: a = 90
         type(a)
Out[13]: int
In [14]: a = 'Clive'
         type(a)
Out[15]: str
```

In Python, variables are free-spirited nomads,they don't tie themselves down to any one type. Unlike the strict and formal Java or C++, where variables must declare their allegiance to a specific type like knights pledging to a king, Python variables are more like fun-loving party guests who wear different outfits depending on the occasion.

So when you write something like:

```
thing = 10
thing = "Now I'm a string!"
thing = [1, 2, 3]
```

Python doesn't bat an eye. It just nods, throws on some sunglasses, and keeps grooving. That's because **Python uses dynamic references**, meaning variable names are just labels that can point to *any* object, and they don't carry type baggage around like some kind of ID badge.

But don't think Python is all fun and games. Oh no,enter **strong typing**. This is Python's inner referee. Even though it lets variables change what they point to, it won't allow incompatible objects to party together unless you clearly state how they should behave.

For example, try to mix a string and an integer:

```
x = "5"
y = 10
print(x + y)  # Boom! TypeError
```

Python immediately throws a flag on the play and says, "Excuse me, sir, this is a type mismatch. Please fix your nonsense." You

have to be explicit about conversions, like calling int(x) or str(y), or Python won't let you continue.

Variables are names for objects within a particular namespace; the type information is stored in the object itself. Some observers might hastily conclude that Python is not a "typed language." This is not true; consider this other example:

```
In [16]: '6' + 6
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)
Input In [18], in <cell line: 1>()
----> 1 '6' + 6
TypeError: can only concatenate str (not "int") to str
```

In some languages, such as Visual Basic, the string '6' might get implicitly converted (or t*ypecasted*) to an integer, thus yielding 12. Yet in other languages, such as JavaScript, the integer 6 might be casted to a string, yielding the concatenated string '66'. In this regard Python is considered a *strongly typed* language, which means that every object has a specific type (or *class*), and implicit conversions will occur only in certain obvious circumstances, such as the following:

```
In [1]: a = 32.54
In [2]: b = 4
# String formatting, to be visited later
In [3]: print('a is {0}, b is {1}'.format(type(a), type(b)))
Out [3]: a is <class 'float'>, b is <class 'int'>
```

Note: The example above returns the data type for variable a and b.

```
In [4]: a / b
Out[4]: 8.135
```

Don't worry though,we'll dive deeper into **typecasting** later in this module and teach you how to convert types like a pro magician

Knowing the type of an object is important, and it's useful to be able to write functions that can handle many different kinds of input. You can check that an object is an instance of a particular type using the isinstance function:

```
In [1]: a = 65
In [2]: isinstance(a, int)
Out[2]: True
```

isinstance can accept a tuple of types if you want to check that an object's type is among those present in the tuple:

```
In [1]: a = 5; b = 4.5
In [2]: isinstance(a, (int, float))
Out[2]: True
In [3]: isinstance(b, (int, float))
Out[3]: True
```

We will discuss more about Tuples and other data structures later in this book.

TYPE CASTING

Type casting (or **type conversion**) is the process of converting a variable from one data type to another.This is like giving your variables a wardrobe change , transforming them from one type to another so they can fit in at different parties (or, well, code blocks).

Python supports two types of type casting:

1. **Implicit Type Casting (Automatic Conversion)**

Python is sometimes smart enough to say,

"Hey, you're mixing an int with a float? No problem, I got this!"

In Implicit type casting, Python automatically converts smaller data types to larger data types to prevent data loss.

Example:

```
x = 5   # Integer
y = 2.5 # Float
result = x + y  # Integer is converted to float automatically
print(result)  # Output: 7.5 (Float)
```

Here, Python automatically upgrades x from an integer to a float so the operation goes smoothly. No errors, no drama. Everyone's happy.

2. **Explicit Type Casting (Manual Conversion)**

Sometimes, Python says,

"Sorry pal, I can't guess what you mean. You gotta tell me exactly how to change this thing."

You have to manually wave your wand (or use functions like int(), float(), str(), etc.) to transform types.

In Explicit type casting, The programmer manually converts a data type using built-in functions. You have to manually wave your wand (or use functions like int(), float(), str(), etc.) to transform types.

Example:

```
a = "100"
b = int(a)  # String to Integer
c = float(b)  # Integer to Float
print(b, c)  # Output: 100 100.0
```

DATES AND TIMES

Python's got you covered when it comes to keeping track of time , no need to shout "What day is it?!" at your laptop like a confused time traveler.

Python provides the **datetime** module to handle date and time operations, including retrieving the current date, formatting timestamps, and performing date calculations.

1. Importing the datetime Module

To work with dates and times, first, import the datetime module:

```
import datetime
```

2. Getting the Current Date and Time

The datetime.now() method returns the current date and time.

```
import datetime
current_time = datetime.datetime.now()
print(current_time)  # Example Output: 2025-02-19 14:30:45.123456
```

To get only the **current date**:

```
today = datetime.date.today()
print(today)  # Example Output: 2025-02-19
```

3. Creating a Specific Date and Time

You can create a custom date using datetime.datetime(year, month, day, hour, minute, second):

```
custom_date = datetime.datetime(2023, 5, 10, 14, 30, 0)
print(custom_date)  # Output: 2023-05-10 14:30:00
```

4. Formatting Dates and Times

Python allows formatting using the .strftime() method:

```
formatted_date = current_time.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_date)  # Example Output: 2025-02-19 14:30:45
```

When you're using Python's strftime() method, it's like handing your datetime object a little outfit and saying, "Go impress the world!"

Here's the list of **format codes** your datetime can wear to look fab:

- **%Y** – The *full-on grown-up* year (e.g., 2025). No abbreviations here.
- **%m** – The month, all properly padded like it's wearing gloves: 01 to 12.

- **%d** – The day of the month, also zero-padded, because style matters: 01 to 31.
- **%H** – The hour, 24-hour clock style, for those who like structure: 00 to 23.
- **%M** – Minutes, because we like to be on time: 00 to 59.
- **%S** – Seconds, for when things get really precise: 00 to 59.

```
from datetime import datetime
now = datetime.now()
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print("Current timestamp:", formatted)
```

5. Parsing Strings into Dates

Ever got a date as a string and Python looks at it like, "Bro, I can't work with this!"?

No worries. We slap on some format magic and boom 💥,Python decodes it like Sherlock with a coffee addiction.

```
import datetime
date_str = "2025-02-19 14:30:00"
parsed_date = datetime.datetime.strptime(date_str, "%Y-%m-%d %H:%M:%S")
print(parsed_date)
```

Convert a string into a `datetime` object using `strptime()`:

```
date_str = "2025-02-19 14:30:00"
parsed_date = datetime.datetime.strptime(date_str, "%Y-%m-%d %H:%M:%S")
print(parsed_date)  # Output: 2025-02-19 14:30:00
```

6. Date Arithmetic (Adding or Subtracting Time)

Use `timedelta` to perform operations on dates:

```
from datetime import datetime, timedelta
future_date = datetime.now() + timedelta(days=7)  # Add 7 days
past_date = datetime.now() - timedelta(weeks=2)   # Subtract 2 weeks
print(future_date, past_date)
```

7. Getting the Current Time

To extract the current time without the date:

```
current_time = datetime.datetime.now().time()
print(current_time)  # Example Output: 14:30:45.123456
```

OPERATORS

Operators in Python are **symbols** that perform operations on variables and values.

With just a flick of the wrist (or a key press), they can do all sorts of things,from math to comparisons to... *magic tricks*. Let's break it down! They are categorized into different types based on their functionality.

1. Arithmetic Operators ***(Used for mathematical calculations)***

Table 2-1: The Arithmetic operators

Operator	Description	Example (a = 10, b = 3)	Output
+	Addition	a + b	13
-	Subtraction	a – b	7
*	Multiplication	a * b	30
/	Division (float)	a / b	3.333
//	Floor Division	a // b	3
%	Modulus (Remainder)	a % b	1
**	Exponentiation	a ** b	1000

2. Comparison (Relational) Operators ***(Used to compare values, return True or False)***

Table 2-2: The Comparison Operations used in Python

Operator	Description	Example (a = 10, b = 3)	Output
= =	Equal to	a = = b	False
!=	Not equal to	a != b	True
>	Greater than	a > b	True
<	Less than	a < b	False
> =	Greater than or equal to	a > = b	True
< =	Less than or equal to	a < = b	False

3. Logical Operators *(Used to combine conditional statement)s*

Table 2-3: The Logical Operators in Python

Operator	Description	Example (x = True, y = False)	Output
And	Returns True if both conditions are True	x and y	False
Or	Returns True if at least one condition is True	x or y	True
Not	Reverses the condition	not x	False

4. Bitwise Operators *(Operate on binary values of numbers)*

Table 2-4: Bitwise Operators in Python

Operator	Description	Example (a = 5 (0101 in binary), b = 3 (0011 in binary))	Output
&	Bitwise AND	a & b	1 (0001)
`	`	Bitwise OR	`a
^	Bitwise	a ^ b	6 (0110)

Operator	Description	Example (a = 5 (0101 in binary), b = 3 (0011 in binary))	Output
	XOR		
~	Bitwise NOT	~a	-6
<<	Left Shift	a << 1	10 (1010)
>>	Right Shift	a >> 1	2 (0010)

5. Assignment Operators ***(Used to assign values to variables)***

Table 2-5: The assignment operators in Python

Operator	Description	Example (x = 5)	Equivalent To
=	Assign value	x = 5	x = 5
+=	Add and assign	x += 3	x = x + 3
-=	Subtract and assign	x -= 2	x = x – 2
*=	Multiply and assign	x *= 4	x = x * 4
/=	Divide and assign	x /= 2	x = x / 2
//=	Floor divide and assign	x //= 2	x = x // 2
%=	Modulus and assign	x %= 3	x = x % 3
**=	Exponentiate and assign	x **= 2	x = x ** 2

6. Membership Operators ***(Check if a value exists in a sequence like a list, tuple, or string)***

Table 2-6: Membership Operators in Python

Operator	Description	Example (x = [1, 2, 3])	Output
In	Returns True if a value exists in	2 in x	True

Operator	Description	Example (x = [1, 2, 3])	Output
	the sequence		
not in	Returns True if a value does NOT exist in the sequence	4 not in x	True

7. Identity Operators *(Check if two variables reference the same object in memory)*

Table 2-7: Identiity Operators in Python

Operator	Description	Example (a = 10, b = 10, c = [1,2], d = [1,2])	Output
Is	Returns True if two variables refer to the same object	a is b	True
is not	Returns True if two variables do not refer to the same object	c is not d	True

Examples:

Let us demonstrate how these operators are used

1. Arithmetic Operators

```
a = 10
b = 3
print(a + b)   # Addition: 13
print(a - b)   # Subtraction: 7
print(a * b)   # Multiplication: 30
print(a / b)   # Division: 3.333
print(a // b)  # Floor Division: 3
print(a % b)   # Modulus: 1
print(a ** b)  # Exponentiation: 1000
```

2. Comparison Operators

```
a = 10
b = 5
print(a == b)  # False
print(a != b)  # True
print(a > b)   # True
print(a < b)   # False
print(a >= b)  # True
print(a <= b)  # False
```

3. Logical Operators

```
x = True
y = False
print(x and y)  # False
print(x or y)   # True
print(not x)    # False
```

4. Bitwise Operators

```
a = 5  # Binary: 0101
b = 3  # Binary: 0011
print(a & b)   # AND: 1  (0001)
print(a | b)   # OR: 7   (0111)
print(a ^ b)   # XOR: 6  (0110)
print(~a)      # NOT: -6
print(a << 1)  # Left Shift: 10 (1010)
print(a >> 1)  # Right Shift: 2  (0010)
```

5. Assignment Operators

```
x = 10
x += 2  # x = x + 2 → 12
x -= 3  # x = x - 3 → 9
x *= 4  # x = x * 4 → 36
x /= 6  # x = x / 6 → 6.0
x //= 2 # x = x // 2 → 3
x %= 2  # x = x % 2 → 1
x **= 3 # x = x ** 3 → 1
print(x)  # Final output: 1
```

6. Membership Operators

```
fruits = ["apple", "banana", "cherry"]
```

```
print("banana" in fruits)  # True
print("grape" not in fruits)  # True
```

7. Identity Operators

```
a = [1, 2, 3]
b = a  # Both refer to the same list
c = [1, 2, 3]  # Different object with the same values
print(a is b)  # True
print(a is not c)  # True
print(a == c)  # True (values are equal)
```

FUNCTIONS

In Python, a **function** is a reusable block of code that performs a specific task. Functions help in organizing code, improving readability, and avoiding repetition. In data science, functions are extensively used for **data manipulation, analysis, visualization, and machine learning model building**.

Python provides two main types of functions:

- **Built-in Functions** (e.g., print(), len(), sum())
- **User-Defined Functions** (custom functions created using the def keyword)

Defining and Calling Functions

To create a function, you first have to **define** it. You use the magical **def** keyword to open up the magic door, followed by the **function name** (think of it like naming your magic potion) and any **parameters** (optional ingredients for your spell): Once you've defined your magical function, you can **call** it anytime you want to make the potion! This is like saying the magic words,*Abracadabra!*

Syntax:

```
def function_name(parameters):
    """Docstring (optional): Describes what the function does."""
    # Function body
```

```
    return result  # (Optional)
```

Example:

```
In [1]: def greet(name):
        """This function prints a greeting message."""
                        return f"Hello, {name}!"

In [2]: print(greet("Clive!"))  # Output: Hello, Clive!
Out [2]: Hello, Clive!
```

In Python, there's no rule that says **only one return statement** is allowed in a function. Nope! You can have **multiple return statements** if your function wants to switch up its game depending on different conditions. If Python doesn't find a return statement, it defaults to returning **None** like a confused wizard who couldn't find his wand

Now, let's dive into the epic battle of **Positional Arguments** and **Keyword Arguments**,it's like a tag-team wrestling match!

1. **Positional Arguments** are like the trusty old **bouncer at the club** . They make sure everyone enters in the right order. When calling a function, you have to respect the order they were defined in:

```
def my_function(x, y, z=7):
```

- Here, x and y are **positional arguments**. They have to be passed in that specific order.

2. **Keyword Arguments** are the **rebels** of the function world. These are the ones that can be **passed in any order** and are often used to provide **default values** or **optional** arguments:

```
def my_function(x, y, z=7):
   print(x, y, z)
```

In the above, z=7 is the **default keyword argument**. If you don't specify z, Python just assumes it's 7. But if you do, you can specify it in any order:

```
my_function(5, 6, z=3)  # I'm giving z a value of 3, just because I can!
```

```
my_function(3.14, 7, 3.5)  # I don't even care about the order!
my_function(10, 20)  # I'm fine with the default value of z=7
```

One of the coolest parts about keyword arguments is that they let you **mix things up**. You don't need to remember the exact order of parameters,just call them by their names. It's like you're speaking a secret code, and Python is all about it!

```
my_function(x=5, y=6, z=7)  # All good, Python!
my_function(y=6, x=5, z=7)  # Yep, I switched them. No problem!
```

You're free to specify keyword arguments in **any order**. It's like ordering pizza and telling the chef, "Hey, I want extra cheese, hold the olives, and no tomatoes! But I don't care when you do it."

Just one simple rule to remember: **Positional arguments must come before keyword arguments**. It's like how you can't have dessert before your main course ! So if you have both, **positional arguments must be listed first**:

```
my_function(5, 6, z=7)  # Fine, it's all in the right order!
my_function(x=5, y=6, z=7)  # Totally cool too, with keyword arguments!
```

Namespaces, Scope, and Local Functions

A **namespace** is a system that manages the names of variables, functions, and objects to avoid conflicts. Python has different types of namespaces:

- **Built-in Namespace** – Contains predefined functions and objects like print(), len(), and abs().
- **Global Namespace** – Stores variables and functions defined at the top level of a script or module.
- **Local Namespace** – Exists within functions and contains variables that are only accessible inside that function.
- **Enclosing Namespace** – Found in nested functions, where an inner function can access variables from an outer function.\

Scope in Python

Scope refers to the visibility of variables within a program. Python follows the **LEGB (Local, Enclosing, Global, Built-in)** rule to determine variable scope:

- **Local Scope** – Variables defined inside a function are only accessible within that function.
- **Enclosing Scope** – Applies to nested functions where an inner function can access variables from an outer function.
- **Global Scope** – Variables defined outside functions are accessible throughout the program unless modified inside a function using global.
- **Built-in Scope** – Includes Python's default functions and libraries.

Example:

```
In [1] x = 10  # Global scope
def outer_function():
    y = 20  # Enclosing scope

    def inner_function():
        z = 30  # Local scope
        print(x, y, z)  # Accessing global, enclosing, and local variables

    inner_function()

outer_function()
```

Local Functions (Nested Functions)

A **local function** is a function defined inside another function. It is useful for encapsulating functionality that should not be accessed outside its parent function.

Example:

```
def outer():
    def inner():
```

```
    return "Hello from inner function!"

  return inner()  # Calling the local function

print(outer())
```

Nested functions are often used in **closures** and **decorators** in Python, making them powerful for organizing and structuring code efficiently.

Application of Functions in Data Science

Functions play a crucial role in **data preprocessing, feature engineering, statistical analysis, and model evaluation**.
We may not be familiar with some libraiers used in this section. However, try to understand how function is used in datascience . We will discuss more about these libraries later.

1. **Functions for Data Manipulation (Pandas)**

Pandas is a key library in data science for handling datasets. Custom functions can be applied to data for transformation.
Example: Applying a function to modify a Pandas DataFrame column:

```
import pandas as pd
# Creating a sample DataFrame
data = {'Name': ['Clive', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data)

# Function to categorize age groups
def age_category(age):
  return "Young" if age < 30 else "Adult"

# Applying the function to the DataFrame
df['Category'] = df['Age'].apply(age_category)
print(df)
```

2. **Functions for Data Analysis (NumPy & Statistics)**

Functions help compute statistical measures like mean, median, and standard deviation using **NumPy**.

Example:

```
import numpy as np

# Sample dataset
data = [12, 15, 14, 10, 18, 20, 22, 25]

# Function to compute basic statistics
def compute_statistics(arr):
    return {
        'Mean': np.mean(arr),
        'Median': np.median(arr),
        'Standard Deviation': np.std(arr)
    }

print(compute_statistics(data))
```

3. **Functions for Data Visualization (Matplotlib & Seaborn)**

Custom functions can be used to automate plotting for data exploration.

Example:

```
import matplotlib.pyplot as plt

# Function to create a bar chart
def plot_bar_chart(categories, values, title="Bar Chart"):
    plt.bar(categories, values, color='skyblue')
    plt.xlabel("Categories")
    plt.ylabel("Values")
    plt.title(title)
    plt.show()

# Sample data
categories = ["A", "B", "C"]
values = [10, 20, 15]
```

```
plot_bar_chart(categories, values)
```

4. Functions in Machine Learning (Scikit-learn)

Functions help in **data preprocessing, feature selection, model training, and evaluation** in machine learning.

Example: Training a Machine Learning Model Using a Function

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Sample dataset
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]])
y = np.array([0, 0, 1, 1, 1])

# Function to train and evaluate a model
def train_model(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    model = LogisticRegression()
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    return accuracy_score(y_test, predictions)

print(f"Model Accuracy: {train_model(X, y):.2f}")
```

Lambda Functions (Anonymous Functions)

A **lambda function** is an anonymous, one-liner function that you can use for quick operations without having to formally define a whole function using def. They're ideal when you need a function for a short time or as an argument for higher-order functions (functions that take other functions as arguments, like map(), filter(), and sorted()).

A **lambda function** is defined using the lambda keyword followed by one or more arguments, a colon, and a single expression. Here's how it looks:

SYNTAX

```
lambda arguments: expression
```

If you need a quick function to add 5 to any number, you can use a **lambda function** like this:

```
add_five = lambda x: x + 5
print(add_five(3))  # Output: 8
```

Example:

```
# Lambda function to square a number
square = lambda x: x ** 2
print(square(5))  # Output: 25
# Using lambda in Pandas
df['Age Squared'] = df['Age'].apply(lambda x: x ** 2)
print(df)
```

Returning Multiple Values

When I first made the leap from Java and C++ to Python, it was like finding out that your favorite cereal can also double as an ice cream topping,**mind-blowing!** One of the coolest tricks Python pulled out of its hat was the ability to return **multiple values** from a function without any fancy, complicated syntax. It's like getting a gift bag full of goodies with a single function call. Here's how it works:

Here's an example:

```
def get_person_info():
    name = "Asuai"
    age = 30
    job = "Data Scientist"
    # Returning multiple values as a tuple
    return name, age, job

# Calling the function and unpacking the values
name, age, job = get_person_info()

print(name)  # Output: Asuai
```

```
print(age)  # Output: 30
print(job)  # Output: Data Scientist
```

In data science (and probably in a few other scientific realms where we wear lab coats and write code in the dark), you'll often find yourself in the magical land of returning multiple values. Remember when we used that cool tuple trick earlier? Well, here's the **behind-the-scenes** look: The function is actually returning just **one object**, but that object is a tuple! Then, Python does a little *sleight of hand* to unpack it into multiple variables.

For example, let's take a step back and see how this looks when we *don't* unpack those values right away:

```
return_value = f()
# In this case, return_value would be a 3-tuple with a, b, and c
```

Now, if you're feeling fancy (and who wouldn't want to be?), you might opt for a more **organized** method of returning values, especially if your values need **labels**. Here's where the **dictionary** comes into play,kind of like putting your values in labeled boxes:

```
def f():
    a = 5
    b = 6
    c = 7
    # Returning a dictionary instead of a tuple
    return {'a': a, 'b': b, 'c': c}

# Call it
return_value = f()
print(return_value)
```

Functions Are Objects

In Python, **functions are first-class objects**, meaning they can be assigned to variables, passed as arguments, returned from other functions, and stored in data structures like lists and dictionaries.

This feature makes Python highly flexible, especially in functional programming and data science.

1. Assigning Functions to Variables

Since functions are objects, they can be assigned to variables and called using the new variable name.

```
def greet(name):
    return f"Hello, {name}!"

greet_function = greet  # Assigning function to a variable
print(greet_function("Alice"))  # Output: Hello, Alice!
```

2 Passing Functions as Arguments

Functions can be passed as arguments to other functions, allowing for high flexibility in programming.

```
def apply_function(func, value):
    return func(value)

def square(x):
    return x ** 2

print(apply_function(square, 5))  # Output: 25
```

3. Returning Functions from Functions

A function can return another function, creating dynamic and reusable code structures.

```
def outer_function():
    def inner_function():
        return "Hello from the inner function!"
    return inner_function  # Returning the function

func = outer_function()  # Assigning the returned function to a variable
print(func())  # Output: Hello from the inner function!
```

ATTRIBUTES AND METHODS

In Python, **attributes** and **methods** are fundamental concepts of object-oriented programming (OOP). They are used to define the properties and behaviors of objects in a class.

1. Attributes (Object Properties)

An **attribute** is a variable that stores data related to an object. Attributes can be:

- **Instance Attributes** – Specific to an object and defined within the __init__ method.
- **Class Attributes** – Shared among all instances of a class and defined outside the __init__ method.

Example:

```
class Car:
    wheels = 4  # Class attribute
    def __init__(self, brand, color):
        self.brand = brand  # Instance attribute
        self.color = color  # Instance attribute
# Creating objects
car1 = Car("Toyota", "Red")
car2 = Car("Honda", "Blue")
print(car1.brand)  # Output: Toyota
print(car2.color)  # Output: Blue
print(Car.wheels)  # Output: 4 (shared among all instances)
```

2. Methods (Object Behaviors)

A **method** is a function that belongs to an object and operates on its attributes. Methods define an object's behavior and are called using the **dot (.) notation**.

- **Instance Methods** – Modify instance attributes and require self.
- **Class Methods** – Work with class attributes and use @classmethod.

- **Static Methods** – Independent of the class and do not use self or cls.

Example:

```
class Car:
    wheels = 4  # Class attribute
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color
    def display_info(self):  # Instance method
        return f"{self.color} {self.brand} with {self.wheels} wheels"
    @classmethod
    def change_wheels(cls, new_wheels):  # Class method
        cls.wheels = new_wheels
    @staticmethod
    def general_info():  # Static method
        return "Cars have engines and wheels."
# Creating an object
car1 = Car("Toyota", "Red")
print(car1.display_info())  # Output: Red Toyota with 4 wheels
Car.change_wheels(6)  # Modifying class attribute
print(car1.display_info())  # Output: Red Toyota with 6 wheels
print(Car.general_info())  # Output: Cars have engines and wheels.
```

- ***Attributes*** *store object-related data.*
- ***Methods*** *define object behavior and operate on attributes.*
- *Python supports* ***instance, class, and static methods*** *for flexible OOP implementation.*
  These concepts are essential for writing modular and reusable code in data science, machine learning, and software development.

Objects in Python typically have both attributes (other Python objects stored "inside" the object) and methods (functions associated with an object that can have access to the object's internal data). Both of them are accessed via the syntax

obj.attribute_name:

```
In [1]: a = 'Clive'
In [2]: a.<Press Tab>
a.capitalize a.format a.isupper a.rindex a.strip a.center a.index a.join a.rjust
a.swapcase a.count a.isalnum a.ljust a.rpartition a.title a.decode a.isalpha a.lower
a.rsplit a.translate a.encode a.isdigit a.lstrip a.rstrip a.upper a.endswith a.islower
a.partition a.split a.zfill a.expandtabs a.isspace a.replace a.splitlines a.find a.istitle
a.rfind a.startswith
```

QUESTIONS

1. Explain how Python manages memory for variables and data types. How does Python handle memory allocation and garbage collection?
2. Python is a dynamically typed language, but type hints (typing module) are used in modern Python programming. Explain the benefits of type hints and provide an example where they improve code readability.
3. Explain the concept of variable scope in Python. What is a closure in Python, and how can it be used effectively in data science applications?
4. Given a long string containing multiple sentences, write a function that finds and returns the longest word in the string. Optimize for performance.
5. Write a function that accepts two numbers and performs division. Ensure the function handles cases where the denominator is zero and returns a proper error message instead of raising an exception.
6. Write a Python function that takes any number of keyword arguments and returns a formatted string of key-value pairs in alphabetical order.
7. Write a Python class that represents a Temperature object. Implement methods to convert temperatures between Celsius, Fahrenheit, and Kelvin.

MODULE 3

CONTROL STRUCTURES

Control structures control the flow of program. In this structure, your Python code learns how to make decisions, repeat itself (intentionally), and generally behave like a clever little robot. This module is all about giving your program some *brains*,with **if-else statements**, **loops**, and **loop control tools** like break and continue. Think of it as teaching your code how to **choose its own adventure** or **do things over and over again without complaining**. Buckle up,Python is about to get a whole lot more logical (and slightly dramatic)! By the end of this module, the reader should have understanding on:

1. **Introduction to Control Structures**
    - Importance of control structures in programming.
    - Overview of different control structures in Python.
2. **Conditional Statements**
    - if, elif, and else statements.
    - Nested conditional statements.
    - Using logical operators (and, or, not) in conditions.
3. **Looping Constructs**
    - for loops and iterating over sequences.
    - while loops and condition-based iteration.
    - Nested loops and their use cases.
4. **Loop Control Statements**
    - break statement to exit loops early.
    - continue statement to skip an iteration.
    - pass statement as a placeholder.

INTRODUCTION

Control structures in Python are like the traffic cops of your code,directing when to stop, go, or take a U-turn! 🚦 They help your program make decisions (if), repeat actions (while, for), and gracefully skip or exit when needed (break, continue, pass). Python's built-in keywords make it easy to write logic that flows smoother than a fresh cup of coffee on a Monday morning. Whether you're building a calculator, a chatbot, or just trying to survive your first coding class,control structures are your go-to tools to keep the code chaos in check.

CONDITIONAL STATEMENTS (DECISION MAKING)

Conditional statements allow the program to execute specific blocks of code based on conditions. Python has three main types of conditional statements

if Statement in Python

The **if statement** is used for decision-making in Python. It allows the program to execute a specific block of code **only if a condition is True.**

Syntax:

If [condition] :

Statement (s)

Example

```
x = 10
if x > 0:
  print("Positive number")
```

Output: Positive number

If-Else Statements

Control structures allow you to make decisions in your code. The if-else statement is used to execute code based on a condition;

it allows the program to execute different blocks of code depending on whether a condition evaluates to True or False.

Syntax:

```
if condition:
 # Code to execute if condition is True
 else:
# Code to execute if condition is False
```

Example

```
age = 18
if age >= 18:
   print("You are an adult.")
else:
   print("You are a minor.")
```

if, elif, and else

The if statement is one of the most well-known control flow statement types. It

checks a condition that, if True, evaluates the code in the block that follows: You can check multiple conditions using elif (short for "else if").

Syntax:

```
if condition1:
   # Code if condition1 is True
elif condition2:
   # Code if condition2 is True
else:
   # Code if all conditions are False
```

Example:

```
x = 10
if x > 0:
   print("Positive number")
elif x == 0:
   print("Zero")
else:
   print("Negative number")
```

Output: Positive number

USING LOGICAL OPERATORS IN DECISION MAKING

By now we should be familiar with the logical operators; and, or, not, covered in module 2. Now we will try to use them in making decisions.

The Logical operators (and, or, not) enhance if statements by allowing multiple conditions to be evaluated at once. Understanding these operators enables developers to create more flexible and efficient decision-making logic in Python.
Logical operators allow us to make decisions based on multiple conditions.

Example 1: **Checking Age and Income for Loan Eligibility**

```
age = 25
income = 50000
if age >= 18 and income >= 30000:
   print("Eligible for loan")
else:
   print("Not eligible")
```

- **Condition 1**: age >= 18 → True
- **Condition 2**: income >= 30000 → True
- **Logical AND** (and) → Both conditions are True, so the person is eligible.

Example 2: **Granting Access Based on Username and Password**

```
username = "admin"
password = "secure123"
if username == "admin" and password == "secure123":
   print("Access granted")
else:
   print("Access denied")
```

Here, both the username and password must match for access to be granted.

Example 3: **Checking Multiple Conditions with** or

```
temperature = 5
raining = True
if temperature < 10 or raining:
   print("Wear a jacket!")
else:
   print("No jacket needed.")
```

- This ensures a jacket is worn if **either** the temperature is low or it is raining.

Example 4: **Using** not **for Boolean Values**

```
logged_in = False
if not logged_in:
   print("Please log in first")
```

- Since logged_in is False, not logged_in becomes True, and the message is printed.

Example 5**: Complex Decision Making**

```
age = 40
has_valid_id = True
criminal_record = False
if (age >= 18 and has_valid_id) and not criminal_record:
   print("You can apply for a driver's license")
else:
   print("You cannot apply")
```

- The person must be **18 or older**, have a **valid ID**, and **no criminal record** to apply.

LOOPING STATEMENTS (ITERATION)

Looping in Python is like putting your code on repeat,because sometimes once just isn't enough! Whether you're cleaning a messy dataset, training a machine learning model 100 times, or just printing "Hello World" until your laptop begs for mercy, loops

have your back. Python gives you two main loop types: for loops (great for going through items like a polite guest at a buffet) and while loops (perfect when you're not sure how long you'll need but you're in it for the long haul). In data science, loops help automate the boring stuff, so you can focus on the cool stats and fancy plots.

Types of loops

1. for loops

for loops are for iterating over a collection (like a list or tuple) . Used when you know the number of iterations (e.g., iterating over a list, array, or DataFrame).

The standard syntax is:

```
for variable in iterable:
    # Code to execute in each iteration
```

```
# Loop through numbers 1 to 5
for i in range(1, 6):
    print("Number:", i)
```

You can advance a for loop to the next iteration, skipping the remainder of the block, using the continue keyword. Consider this code, which sums up integers in a list and skips None values:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
if value is None:
continue
total + = value
```

2. while loops

A while loop specifies a condition and a block of code that is to be executed until the condition evaluates to False or the loop is explicitly ended with break. It is useful when working with streaming data or performing operations until a condition is met.

Syntax:

```
while condition:
  # Code to execute in each iteration
```

Example:

```
num = 1
while num < = 5:
  print(num)
  num + = 1
```

LOOP CONTROL STATEMENTS IN PYTHON

Loop control statements **alter the flow of loops** (e.g., for and while). Think of loop control statements as traffic signals inside your loops,they tell your code when to keep going, when to skip a turn, and when to slam the brakes and stop entirely. Python offers three main loop control superheroes:

1. **break** – The "I'm outta here!" statement. It ends the loop early, like storming out of a boring meeting before it's over.
2. **continue** – The "skip this one!" statement. It politely tells the loop to skip the current iteration and jump to the next.
3. **pass** – The "nothing to see here" statement. It's a placeholder that lets your code say "I'll deal with this later."

1. break **Statement** *(Stops the Loop)*

The break statement **immediately exits the loop**, even if the condition is still True.

Syntax:

```
for variable in iterable:
  if condition:
    break  # Exit the loop
```

Example 1: break **in a** for **Loop**

```
for num in range(10):
    if num == 5:
        break  # Stops when num is 5
    print(num)
```

Output:

```
0
1
2
3
4
```

Example 2: break **in a** while **Loop**

```
count = 0
while count < 10:
    if count == 3:
        break  # Exits when count reaches 3
    print(count)
    count += 1
```

Output:

```
0
1
2
```

2. continue **Statement** *(Skips Current Iteration)*

The continue statement **skips the current iteration** and moves to the next loop cycle.

Syntax:

```
for variable in iterable:
    if condition:
        continue  # Skip this iteration
```

Example 1: continue **in a** for **Loop**

```
for num in range(5):
    if num == 2:
        continue  # Skips when num is 2
```

```
    print(num)
```

Output:

```
0
1
3
4
```

(2 is skipped, but the loop continues.)

Example 2: continue **in a** while **Loop**

```
num = 0
while num < 5:
    num += 1
    if num == 3:
        continue  # Skips when num is 3
    print(num)
```

Output:

```
1
2
4
5
```

(3 is skipped.)

3. pass **Statement** *(Does Nothing – Placeholder)*

The pass statement is a **placeholder** that allows you to write syntactically correct code without executing anything. It is useful when you need to define a structure but don't want to implement logic yet.

Synatx:

```
if condition:
    pass  # Placeholder, does nothing
```

The pass statement **does nothing** and is used as a placeholder for future code.

Syntax:

```
if condition:
    pass  # Placeholder, no action taken
```

Example 1: pass **in a** for **Loop**

```
for num in range(5):
    if num == 2:
        pass  # Does nothing
    print(num)
```

Output:

```
0
1
2
3
4
```

(No effect, just a placeholder.)

Example:

```
x = 10
if x > 5:
    pass  # Placeholder, no action yet
else:
    print("x is 5 or less")
for i in range(5):
    pass  # Placeholder for future logic
print("Loop completed")
```

Example 2: pass **in a Function**

```
def process_data():
    pass  # Placeholder for function logic
print("Function defined but not implemented.")
```

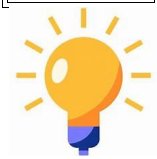
Output:

```
Function defined but not implemented.
```

Table 3-1: Key Differences in loop control

Statement	Purpose	Effect
Break	Stops the loop completely	Loop terminates immediately
Continue	Skips the current	Moves to the next iteration

Statement	Purpose	Effect
	iteration	
Pass	Placeholder (does nothing)	Code runs without effect



Practical Use Cases in Data Science

✔ **break** – *Stop looping when a certain condition is met (e.g., early stopping in ML).*

✔ **continue** – *Skip processing invalid/missing data while iterating.*

✔ **pass** – *Reserve space for future logic in loops, functions, or classes.*

RANGE FUNCTION

We have been using range in this module without understanding how it works. Now let's dive into the range() in-built function.

The range() function is used to generate a sequence of numbers. It is commonly used in loops to control iteration.

Syntax

range(start, stop, step)

- *start (optional) – The first number in the sequence (default is 0).*
- *stop (required) – The number where the sequence stops (**not included**).*
- *step (optional) – The difference between consecutive numbers (default is 1).*

Example:

```
for i in range(5):
```

```
    print(i)
Start, Stop
for i in range(2, 6):
    print(i)
Start, Stop, Step
for i in range(1, 10, 2):
    print(i)
```

Steps can be assigned negative values. If step is negative, the sequence counts **backward**.

```
for i in range(10, 0, -2):
    print(i)
```

In Data science, the range() function can be used in developing a list. For example, We will generate a list of numbers , 0 to 5 (excluding the number 5).

```
numbers = list(range(5))
print(numbers)
```

This is another way to generate a List as explained earlier.

```
In [1]: range(10)
Out[1]: range(0, 10)
In [2]: list(range(10))
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

TERNARY EXPRESSIONS

(CONDITIONAL EXPRESSION)

A **ternary expression** (or **conditional expression**) in Python is a one-liner if-else statement that returns a value based on a condition. It allows you to combine an if-else block that produces a value into a single line or expression.

Syntax of Ternary Expression

Value if true if condition else value_if_false

- If the condition is **True**, value_if_true is returned.
- If the condition is **False**, value_if_false is returned.

Examples of Ternary Expressions

Example 1: Basic Ternary Expression

```
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)
```

Output:

Adult

(Since age >= 18 *is True,* "Adult" *is assigned to* status*.)*

Example 2: Finding the Maximum of Two Numbers

```
a, b = 10, 20
max_value = a if a > b else b
print(max_value)
```

Output:

20

Example 3: Even or Odd Check

```
num = 7
result = "Even" if num % 2 == 0 else "Odd"
print(result)
```

Output:

Odd

Example 4: Nested Ternary Expression

```
x = -5
result = "Positive" if x > 0 else "Zero" if x == 0 else "Negative"
print(result)
```

Output:

Negative

(Since x *is* -5*, it falls into the last condition.)*

In Data science, Teneray expression can be used in:

✔ *Assigning labels based on conditions (e.g., classification tasks)*

✔ *Feature engineering (e.g., normalizing data)*

✔ *Applying transformations in a concise way*

Example: Assigning a Label in a Pandas DataFrame

```
import pandas as pd
df = pd.DataFrame({'Age': [15, 25, 17, 30]})
df['Category'] = df['Age'].apply(lambda x: "Adult" if x >= 18 else "Minor")
print(df)
```

Output:

```
   Age Category
0   15   Minor
1   25   Adult
2   17   Minor
3   30   Adult
```

QUESTIONS

1. How do control structures improve the efficiency of a program?
2. How does the elif statement differ from if and else?
3. What are logical operators, and how are they used in conditional statements?
4. What will be the output of the following code?

```
x = 10
if x > 5 and x < 15:
   print("x is in range")
else:
   print("x is out of range")
```

5. What is an infinite loop, and how can it be avoided?
6. How is the pass statement different from break and continue?
7. What will be the output of the following code?

```
for i in range(5):
   if i == 3:
      break
   print(i)
```

8. How can you use a loop control statement to skip even numbers in a loop from 1 to 10?
9. Write a code that sums all numbers from 0 to 99,999 that are multiples of 3 or 5:
10. Write a code that compares two numbers and returns the minimum number (using the Tenary expression)
11. Consider Table 3-2, write a program that grades the student of a University based on their examination scores

Table 3-2: Grading scale for University students.

S/N	SCORES	GRADES
1	0-39	F
2	40-45	D
3	46-55	C
4	56-69	B
5	70-100 (>=70)	A
6	>100	OUT-OF-RANGE
7	<0	NEGATIVE VALUE

Use the if-elif-else statement

MODULE 4

INTRODUCTION TO PYTHON LIBRARIES FOR DATA SCIENCE

Python libraries are like a cheat code for programmers: they hand you powerful tools on a silver platter so you can spend less time reinventing the wheel and more time doing the fun, brainy stuff. Want to wrestle with massive datasets? There's a library for that. Need to crunch numbers like a caffeinated accountant? There's a library for that too. Dreaming of plotting jaw-dropping graphs that make your friends go "wow"? Yup, libraries got your back!
By the end of this module, readers will:

1. Understand the concept of Python libraries and their importance in data science.
2. Learn how Python libraries simplify data analysis, visualization, and machine learning tasks.
3. Gain knowledge of the different types of Python libraries used in data science.
4. Learn how to import libraries using Python's import statement.
5. Recognize the role of libraries in enhancing efficiency and reducing repetitive coding.

INTRODUCTION

Python has a rich ecosystem of libraries for data science. These libraries play a crucial role in making programming more efficient by providing pre-built functions and tools that eliminate the need to write complex code from scratch. Instead of manually implementing algorithms for data manipulation, mathematical operations, or machine learning, developers can use optimized

functions from libraries like **NumPy**, **pandas**, and **scikit-learn**. This not only saves time but also reduces the chances of errors. Libraries ensure consistency by offering standardized implementations, making code more readable, maintainable, and reusable. Additionally, they optimize performance by leveraging efficient, low-level operations (e.g., C/C++ backend in NumPy). By using libraries, programmers can focus on solving problems rather than reinventing basic functionalities, ultimately accelerating development and improving productivity. Some of the libraries required for datascience are:

Think of Python libraries as your super-smart, overachieving best friends who always have the answers , and they *never* complain. Instead of you sweating over writing thousands of lines of complicated code, these libraries show up like, "Relax, buddy, we got this!" Whether it's crunching numbers faster than a caffeine-fueled squirrel (thanks, NumPy), or taming wild, messy datasets like a data-wrangling cowboy (thank you, pandas), these libraries do the heavy lifting while you look like a genius. Honestly, using Python without libraries is like trying to dig a swimming pool with a spoon , why suffer when you have power tools?

NumPy (NUMERIC PYTHON)

NumPy, an abbreviation for **Numerical Python**, is a foundational library in the Python ecosystem for high-performance numerical computing. It provides powerful multi-dimensional array objects, along with a suite of sophisticated functions and tools for performing complex mathematical operations efficiently.

Beyond simple array handling, NumPy underpins the core data structures and algorithmic frameworks required for scientific and engineering applications in Python. Acting as the "connective tissue" between Python and highly optimized C/C++ routines,

NumPy dramatically accelerates computation while maintaining the simplicity and readability of Python code.
In many ways, it is difficult to overstate NumPy's importance , it is not merely a library but the computational backbone of modern data science, machine learning, and scientific research in Python.

Elements of NumPy

• A fast and efficient multidimensional array object *ndarray*
• Functions for performing element-wise computations with arrays or mathematical operations between arrays
• Tools for reading and writing array-based datasets to disk
• Linear algebra operations, Fourier transform, and random number generation
• A mature C API to enable Python extensions and native C or C++ code to access NumPy's data structures and computational facilities
Beyond the fast array-processing capabilities that NumPy adds to Python, one of its primary uses in data science is as a container for data to be passed between algorithms and libraries. For numerical data, NumPy arrays are more efficient for storing and manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying data into some other memory representation.
That's why most serious number-crunching tools in Python either worship the NumPy array like a sacred relic or at least make sure they play nice with it. Long story short: if you're doing numerical computing in Python, you're basically living in NumPy's world , you're just paying rent.

Importing the NumPy Libfary

By convention, np is used as an alias for NumPy while importing. The syntax of importing NumPy library:

Import numpy as np

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr.mean())  # Output: 3.0
```

Note: The numpy library will be explained in detailed later.

PANDAS (PANEL DATA)

Pandas is used for data manipulation and analysis. pandas provides high-level data structures and functions designed to make working with structured or tabular data fast, easy, and expressive. Since its emergence in 2010, it has helped enable Python to be a powerful and productive data analysis environment. The primary objects in pandas that will be used in this book are the DataFrame, a tabular, column-oriented data structure with both row and column labels, and the Series, a one-dimensional labeled array object. Pandas blends the high-performance, array-computing ideas of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. Since data manipulation, preparation, and cleaning is such an important skill in data science,

For users of the R language for statistical computing, the DataFrame name will be familiar, as the object was named after the similar R data.frame object. Unlike Python, data frames are built into the R programming language and its standard library. As a result, many features found in pandas are typically either part of the R core implementation or provided by add-on packages.

Importiing the pandas library

By convention, pd is used as an alias for pandas while importing. The syntax of importing the pandas library:

```
Import pandas as pd
```

Let's demosrtate how to create a dataframe in pandas.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
print(df)
```

Matplotlib

Matplotlib is a popular Python library used for data visualization. It provides a variety of plotting functions to create static, animated, and interactive visualizations. The library is highly customizable and supports plots like line graphs, bar charts, histograms, scatter plots, and more. The pyplot module in Matplotlib offers a MATLAB-like interface for easy plotting. It integrates well with libraries like NumPy and Pandas, making it useful for data science and machine learning applications.

In short, **Matplotlib** is like the artsy friend who can turn even the most boring spreadsheet into a jaw-dropping gallery. Want a simple line graph? Done. Need a scatter plot that looks like it belongs in a modern art museum? Easy. Dreaming of a bar chart so beautiful it deserves its own

Instagram account? Matplotlib says, "Hold my coffee."

And the best part? It's ridiculously flexible , you can tweak, stretch, and color your plots until they look *just right* (or until you've completely forgotten what you were originally analyzing... oops). With Matplotlib by your side, your data doesn't just speak , it throws a full-blown musical concert!

Importing matplolib library and the pyplot library

By convention, mp and plt are used as aliases for matplot library and pyplot module respectively while importing. The syntax of importing matplot.pyplot::

Import matplotlib.pyplot as plt

Example

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

SEABORN

Seaborn is a Python data visualization library built on top of Matplotlib. It provides a high-level interface for creating attractive and informative statistical graphics. Seaborn simplifies complex visualizations with functions for drawing histograms, scatter plots, box plots, violin plots, and heatmaps. It integrates well with Pandas, allowing for easy visualization of DataFrame-based data. Seaborn also supports theme customization and statistical analysis, making it useful for exploratory data analysis and machine learning.

Think of **Seaborn** as Matplotlib's cooler, better-dressed cousin who shows up to the party and immediately steals the spotlight. While Matplotlib gives you the raw tools to make a plot, Seaborn hands you a masterpiece on a silver platter , color-coordinated, beautifully styled, and ready for Instagram.

Want a heatmap so gorgeous it makes your CPU sweat? A violin plot so elegant it could play at a royal wedding? Seaborn's got you covered. Plus, it plays super nicely with Pandas, so you can throw a messy DataFrame at it, and Seaborn will somehow turn it into data art. With Seaborn, your exploratory data analysis isn't just smart , it's stunning.

Importing Seaborn Library

By convention, sns is used as an alias for Seaborn while importing. The syntax of importing the seaborn library:

Import seaborn as sns

Example

```
import seaborn as sns
sns.histplot([1, 2, 2, 3, 3, 3])
plt.show()
```

SCIKIT-LEARN

Scikit-learn is used for machine learning. Scikit-learn is a machine learning library built from NumPy, SciPy, and Matplotlib. Scikit-learn offers simple and efficient tools for common tasks in data analysis and data science such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

It includes submodules for such models as:

- Classification: SVM, nearest neighbors, random forest, logistic regression, etc.
- Regression: Lasso, ridge regression, etc.
- Clustering: *k*-means, spectral clustering, etc.
- Dimensionality reduction: PCA, feature selection, matrix factorization, etc.
- Model selection: Grid search, cross-validation, metrics
- Preprocessing: Feature extraction, normalization

Along with pandas, statsmodels, and IPython, scikit-learn has been critical for enabling Python to be a productive data science programming language. To use Scikit-learn effectively, it is often integrated with other libraries in the data science ecosystem.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

SCIPY LIBRARY

SciPy SciPy (Scientific Python) is an open-source library built on **NumPy** that provides advanced mathematical, scientific, and engineering functions. It is widely used in data science, machine learning, and scientific computing for tasks like optimization, signal processing, and statistical analysis.

Key Elements of SciPy

1. **Optimization** – Solving minimization and root-finding problems.
2. **Linear Algebra** – Matrix operations and solving linear equations.
3. **Statistics** – Probability distributions, hypothesis testing, and descriptive statistics.
4. **Signal Processing** – Fourier transforms, filtering, and convolution.
5. **Interpolation** – Estimating new data points from existing ones.
6. **Integration** – Numerical integration and solving differential equations.

Leveraging SciPy in Data Science

1. Optimization (Finding Minimum or Maximum of a Function)

Used in machine learning for hyperparameter tuning and cost function optimization.

```
from scipy.optimize import minimize
# Define a simple function
def f(x):
    return x**2 + 5*x + 6

# Find the minimum of the function
```

```
result = minimize(f, x0=0)  # x0 is the initial guess
print("Minimum:", result.x)
```

2. Linear Algebra (Solving Systems of Equations)

SciPy provides efficient matrix operations.

```
import numpy as np
from scipy.linalg import solve

A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
x = solve(A, b)  # Solves Ax = b
print("Solution:", x)
```

3. Statistical Analysis (Descriptive Statistics and Hypothesis Testing)

SciPy extends NumPy's statistical functions.

```
from scipy import stats
data = [2, 8, 0, 4, 1, 9, 7, 3]
mean = np.mean(data)
std_dev = np.std(data)
t_stat, p_value = stats.ttest_1samp(data, popmean=5)  # Hypothesis test
print("Mean:", mean, "| Std Dev:", std_dev, "| P-value:", p_value)
```

4. Signal Processing (Fourier Transform & Filtering)

Useful for analyzing time-series and signal data.

```
from scipy.fft import fft
import matplotlib.pyplot as plt

signal = np.sin(2 * np.pi * np.linspace(0, 1, 100))  # Example signal
transformed = fft(signal)
plt.plot(abs(transformed))  # Plot frequency spectrum
plt.show()
```

5. Numerical Integration (Solving Differential Equations)

Used in physics, engineering, and ML models like neural ODEs.

```
from scipy.integrate import quad
# Define function to integrate
def func(x):
```

```
    return x**2

result, _ = quad(func, 0, 5)  # Integrate from 0 to 5
print("Integral result:", result)
```

STATSMODELS

Statsmodels is a Python library for **statistical modeling, hypothesis testing, and econometrics**. It provides advanced statistical tools for analyzing relationships between variables, making it a powerful alternative to Scikit-learn for regression and time-series analysis.

Key Elements of Statsmodels

1. **Regression Analysis** – Linear, logistic, and generalized linear models.
2. **Time-Series Analysis** – AR, ARMA, and ARIMA models for forecasting.
3. **Hypothesis Testing** – T-tests, ANOVA, and chi-square tests.
4. **Statistical Distributions** – Probability distributions and density estimation.
5. **Robust Models** – Nonparametric regression and generalized estimating equations (GEE).

Leveraging Statsmodels in Data Science

1. Linear Regression (Ordinary Least Squares - OLS)

Statsmodels provides detailed statistical summaries.

```
import statsmodels.api as sm
import numpy as np

# Sample data
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])
X = sm.add_constant(X)  # Add intercept term
```

```
model = sm.OLS(y, X).fit()  # Fit OLS regression
print(model.summary())  # Detailed statistical output
```

2. Logistic Regression (Binary Classification)

Useful for classification problems.

```
import statsmodels.api as sm
import numpy as np
# Example data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([0, 0, 1, 1, 1])  # Binary target
X = sm.add_constant(X)  # Add intercept
model = sm.Logit(y, X).fit()
print(model.summary())
```

3. Time-Series Forecasting (ARIMA Model)

Statsmodels supports ARIMA for time-series predictions.

```
import statsmodels.api as sm
import pandas as pd
# Example time-series data
data = pd.Series([112, 118, 132, 129, 121, 135, 148, 148, 136, 119])
# Fit ARIMA model
model = sm.tsa.ARIMA(data, order=(2, 1, 2)).fit()
print(model.summary())

# Predict next values
forecast = model.forecast(steps=5)
print("Forecast:", forecast)
```

4. Hypothesis Testing (T-Test Example)

Compare means of two samples.

```
import statsmodels.stats.weightstats as smw
import numpy as np

group1 = np.array([12, 15, 14, 10, 9, 11])
group2 = np.array([20, 22, 19, 18, 25, 23])

t_stat, p_value, _ = smw.ttest_ind(group1, group2)
print("T-Statistic:", t_stat, "| P-Value:", p_value)
```

HANDLING MISSING LIBRARIES USING PIP (PYTHON PACKAGE INSTALLER)

You may encounter an error if you import a library that does not exist on your machine. You need to install a Python library using pip when a required library is not included in Python's standard library, when you encounter a ModuleNotFoundError, or when setting up a new environment. It is also necessary when upgrading a library to access new features or fixing compatibility issues by installing a specific version.

Pip is the standard tool for installing Python libraries. To install a new library, use the following command in the terminal or command prompt:

```
pip install library name
```

For example, to install NumPy, run:

```
pip install numpy
```

If you need to install a specific version, use:

```
pip install numpy==1.21.0
```

To upgrade an existing library:

```
pip install --upgrade numpy
```

To install multiple libraries at once, list them in a text file (requirements.txt) and run:

```
pip install -r requirements.txt
```

This ensures your Python environment has the necessary libraries for data science and development.

QUESTIONS

1. Why are libraries important in data science?
2. How do libraries improve efficiency in programming?
3. What is the difference between a library and a module in Python?
4. What happens if you try to import a library that is not installed?
5. What is the difference between import numpy and from numpy import array?
6. Explain the use of as in the statement import pandas as pd.
7. Why is numpy preferred for numerical computations over standard Python lists?
8. What is the difference between matplotlib and seaborn for data visualization?
9. What is scikit-learn used for in data science?

MODULE 5

FILE HANDLING IN PYTHON FOR DATA SCIENCE

Working with files (especially CSV, Excel documents) is a very crucial aspect in datascience. It is imperative that we understand how these are handled.

Think of files like the treasure chests of data science , hidden away, locked up, and sometimes a little dusty. Whether it's a squeaky clean CSV or a grumpy old Excel file full of weird formatting decisions, knowing how to open, read, and manipulate these files is like having the keys to the kingdom. Without proper file handling skills, you're basically a pirate without a map , lots of ambition, but nowhere to sail!

Mastering file handling means you can finally stop fearing the "File Not Found" error like it's a horror movie jump scare , and start confidently pulling in data like a boss.

In this module, readers will:

1. **Understand File Handling Basics** – Learn how to open, read, write, and append to files in Python.
2. **Work with Different File Formats** – Explore handling **CSV** and **JSON** files, which are commonly used in data science.
3. **Learn File Closing Best Practices** – Understand the importance of closing files to prevent resource leaks.
4. **Integrate File Handling with Data Science Workflows** – Learn how to load and store data efficiently in Python's data science ecosystem.
5. **Handle Missing Values in Pandas** – Discover techniques for identifying and managing missing data in datasets.

6. **Perform File Parsing** – Learn how to extract and process structured data from different file formats.

By the end of this module, readers will be able to efficiently handle file operations and integrate data into their data science projects.

INTRODUCTION

File handling is essential in data science for reading datasets, storing processed data, and managing large-scale data pipelines. Python provides built-in functions and libraries like pandas to handle various file formats efficiently.

OPENING FILES IN PYTHON

In Python, opening files is a common operation performed using the built-in open() function. This function creates a file object, allowing you to read, write, or modify the file. The basic syntax is

file = open(filename, mode)

where filename is the file's name (including path if needed) and mode specifies the operation (e.g., 'r' for read, 'w' for write, 'a' for append, or 'r+' for read and write). Python also supports binary ('b') and text ('t') modes.

To ensure proper resource management, it's recommended to use files within a with block, which automatically closes the file after operations are done. For example:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Common File Modes:

In Python, when working with files, you need to specify the **mode** in which the file should be opened. The file mode

determines whether you want to read, write, append, or perform other operations on the file.

Table 5-1: Common file modes

Mode	Description
'r'	Read (default mode, file must exist)
'w'	Write (creates a file if it doesn't exist, truncates if it does)
'a'	Append (adds content to an existing file)
'b'	Binary mode (used with other modes, e.g., 'rb' for reading binary files)
'x'	Exclusive creation (fails if file exists)
't'	Text mode (default mode, used with other modes)

Example:

```
file = open("data.txt", "r")
file.close()
```

The statement file = open("data.txt", "r") opens the file **data.txt** in **read mode ("r")**, meaning the file must exist, and its contents can be read but not modified. If the file does not exist, Python raises a **FileNotFoundError**. The variable **file** becomes a file object, allowing operations like .read(), .readline(), or .readlines() to access its contents. After reading, it's essential to close the file using file.close() to free system resources.

READING FILES

In Python, reading files is a common operation performed using the open() function with the 'r' (read) mode. The simplest way is to use file.read() to fetch the entire content as a string, or file.readlines() to get a list of lines. For memory efficiency with large files, looping through the file object line by line is preferred.

1. read() Method

Reads the entire file content as a string.

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

2. readline() Method

Reads a single line at a time.

```
file = open("data.txt", "r")
line = file.readline()
print(line)
file.close()
```

3. readlines() Method

Reads all lines and returns a list.

```
file = open("data.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

WRITING TO FILES

In Python, writing to files is done using the open() function with modes like 'w' (write) or 'a' (append). The 'w' mode overwrites the file if it exists, while 'a' adds content to the end without deleting existing data. The write() method is used to insert text, and writelines() can write a list of strings.

1. write() Method

Writes a string to a file.

```
file = open("output.txt", "w")
file.write("Hello, Data Science!")
file.close()
```

For safety, files should be opened in a with block to ensure proper closing.

Example

```
# Writing to a file (overwrites existing content)
with open('example.txt', 'w') as file:
```

```
    file.write("Hello, World!\n")
    file.write("This is a new line.")
```

2. writelines() Method

Writes a list of strings to a file.

```
lines = ["Line 1\n", "Line 2\n"]
file = open("output.txt", "w")
file.writelines(lines)
file.close()
```

For safety, files should be opened in a with block to ensure proper closing.

```
lines = ["First line\n", "Second line\n", "Third line\n"]
with open('example.txt', 'w') as file:
    file.writelines(lines)
```

APPENDING TO FILES

Appending allows you to add new content to the end of an existing file without overwriting it. In Python, this is done by opening the file in `'a'` **(append) mode** using the `open()` function.

```
file = open("output.txt", "a")
file.write("New Data\n")
file.close()
```

For safety, files should be opened in a with block to ensure proper closing.

```
with open("example.txt", "a") as file:
    file.write("\nAppending new line.")  # Adds content without deleting existing data
```

WORKING WITH CSV FILES

CSV files are commonly used in data science. The `csv` module and `pandas` library provide functionality for handling CSV files.
Using csv Module

```
import csv
with open("data.csv", "r") as file:
```

```
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Using pandas

```
import pandas as pd
df = pd.read_csv("data.csv")
print(df.head())
note: df.head prints the first 5 rows in the dataset/CSV file.
```

HANDLING JSON FILES

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy to read and write. JSON is widely used for data storage and exchange.

Python provides the built-in json module to work with JSON data.

Reading JSON

Use json.load() to read JSON data into a Python object.

```
import json
with open("data.json", "r") as file:
    data = json.load(file)
print(data)
```

Writing JSON

Use json.dump() to write a Python object (e.g., dict, list) to a JSON file.

```
with open("output.json", "w") as file:
    json.dump(data, file, indent=4)
```

Converting JSON Strings to Python Objects

Use json.loads() to parse a JSON string.

```
json_str = '{"name": "Bob", "age": 30}'
python_dict = json.loads(json_str)
```

Converting Python Objects to JSON Strings

Use json.dumps() to serialize a Python object into a JSON string.

```
data = {"name": "Charlie", "age": 35}
```

```
json_str = json.dumps(data, indent=4)
print(json_str)
```

CLOSING FILES

Properly closing files is essential to free up system resources and prevent data corruption. Python provides multiple ways to ensure files are closed correctly.

1. Using with Statement (Recommended)

The safest way to handle files is using the with statement, which automatically closes the file after the block executes,even if an error occurs.

```
with open("example.txt", "r") as file:
    content = file.read()
# File is automatically closed here
```

2. Manual Closing with close()

If not using with, you must explicitly call file.close() to release resources.

```
file = open("example.txt", "r")
try:
    content = file.read()
finally:
    file.close()  # Ensures file is closed even if an error occurs
```

Risks of Not Closing Files in Python

Failing to properly close files in Python can lead to several serious issues, affecting both program performance and data integrity. When a file is opened but not closed, the operating system retains control of the file handle, consuming system resources unnecessarily. Over time, if multiple files remain open,especially in long-running programs,this can exhaust available file descriptors, leading to errors like "Too many open files", which can crash the application.

Additionally, Python buffers (temporarily holds) data when writing to files for efficiency. If a program terminates unexpectedly or a file is not closed, buffered data may never be written to disk, resulting in partial or corrupted files. This is particularly dangerous for critical operations like database transactions or log file updates. Another risk involves file locks: an open file may prevent other processes (or even the same program) from accessing or modifying it, causing conflicts in multi-threaded or distributed systems.

While Python's garbage collector eventually closes unreferenced file objects, relying on this is unsafe,it may not happen immediately, and exceptions can bypass cleanup. The best practice is to always use a with block or explicitly call close() in a finally clause to ensure resources are released promptly and data is saved correctly.

Table 5-2: Summary Table of File Methods

Method	Description
open(filename, mode)	Opens a file in the specified mode
read()	Reads the entire file as a string
readline()	Reads a single line from the file
readlines()	Reads all lines and returns a list
write(string)	Writes a string to a file
writelines(list)	Writes multiple lines from a list
close()	Closes the file
csv.reader(file)	Reads CSV data into lists
csv.writer(file)	Writes data to a CSV file
json.load(file)	Loads JSON data from a file
json.dump(data, file)	Writes JSON data to a file

INTEGRATING THE BASIC DATA SCIENCE ECOSYSTEM IN DATA LOADING, STORAGE.

Accessing data is a necessary first step for using most of the tools in this book. I'm going to be focused on data input and output using pandas, though there are numerous tools in other libraries to help with reading and writing data in various formats. Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. Table 5-3 summarizes some of them, though read_csv and read_table are likely the ones you'll use the most.

Table 5-3: Functions for reading tabular data

Function	Description
read_csv	Load delimited data from a file, URL, or file-like object; uses a comma as the default delimiter.
read_table	Load delimited data from a file, URL, or file-like object; uses a tab ('\t') as the default delimiter.
read_fwf	Read data in fixed-width column format (i.e., no delimiters).
read_clipboard	Version of read_table that reads data from the clipboard; useful for converting tables from web pages.
read_excel	Read tabular data from an Excel **XLS or XLSX** file.
read_hdf	Read **HDF5** files written by pandas.
read_html	Read all tables found in the given **HTML** document.
read_json	Read data from a **JSON** (JavaScript Object Notation)

Function	Description
	string representation.
read_msgpack	Read pandas data encoded using the **MessagePack** binary format.
read_pickle	Read an arbitrary object stored in **Python pickle** format.
read_sas	Read a **SAS dataset** stored in one of the SAS system's custom storage formats.
read_sql	Read the results of a **SQL query** (using SQLAlchemy) as a pandas DataFrame.
read_stata	Read a dataset from **Stata** file format.
read_feather	Read the **Feather** binary file format.

These functions help **convert text data into a DataFrame**. Each function has optional arguments that allow customization, such as specifying delimiters, handling missing values, and defining data types.

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

Examples

```
In [1]: pd.read_csv('folder1/ex2.csv', header=None)
Out[1]:
0 1 2 3 4
0 1 2 3 4 hello
1 5 6 7 8 world
2 9 10 11 12 foo

In[2]: pd.read_csv('folder1/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[2]:
a b c d message
0 1 2 3 4 hello
1 5 6 7 8 world
```

```
2 9 10 11 12 foo
```

HANDLING MISSING VALUES IN PANDAS FILE PARSING

Handling missing values is a crucial aspect of file parsing in **pandas**. Missing data typically appears as either an **empty string** or a predefined **sentinel value** (e.g., "NA", "NULL"). By default, pandas recognizes common missing value indicators such as **"NA"** and **"NULL"** and converts them into **NaN (Not a Number)**.

Example: Handling Missing Data in CSV Files

1. Default Missing Value Handling

Given the following **CSV file (examples/ex5.csv):**

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
```

When reading this file using read_csv():

```
import pandas as pd
result = pd.read_csv('examples/ex5.csv')
print(result)
Output:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Here, **"NA"** was automatically converted to **NaN**, and the missing value in column c was also treated as **NaN**.

2. Checking for Missing Values

To check where missing values exist:

```
pd.isnull(result)
```

Output:

```
  something     a     b     c     d  message
0     False  False  False  False  False     True
1     False  False  False   True  False    False
2     False  False  False  False  False    False
```

- True indicates missing values (NaN).
- False means the value is present.

3. Custom Missing Value Identifiers

If a dataset uses different markers for missing values (e.g., "NULL"), you can explicitly specify them using the **na_values** argument:

```
result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])
print(result)
```

This ensures that "NULL" is treated as a missing value.

4. Column-Specific Missing Value Handling

You can define different missing value indicators for different columns using a **dictionary**:

```
sentinels = {'message': ['foo', 'NA'], 'something': ['two']}
result = pd.read_csv('examples/ex5.csv', na_values=sentinels)
print(result)
```

Output:

```
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

- "foo" and "NA" in the "message" column are treated as **NaN**.
- "two" in the "something" column is treated as **NaN**.

5. **Key** read_csv **and** read_table **Arguments**

Table 5-4 summarizes important parameters for handling file parsing in pandas:

Argument	**Description**
Path	Path to the file, URL, or file-like object.
sep / delimiter	Character or regex to split fields (default: , for read_csv, \t for read_table).
Header	Row number for column names (default: 0). Use None if no header row.
index_col	Column(s) to set as index.
Names	List of column names (use with header=None).
Skiprows	Number of rows to skip at the beginning.
na_values	List or dict of missing value indicators.
Comment	Character(s) indicating comments in the file.
parse_dates	Try parsing columns as datetime.
converters	Dictionary of column-specific conversion functions.
Nrows	Number of rows to read.
Iterator	Return a **TextParser** object for batch reading.
chunksize	Number of rows per chunk (for iteration).
encoding	Text encoding (e.g., "utf-8").
Squeeze	If the result has one column, return a **Series** instead of a **DataFrame**.
Thousands	Character for thousands separator (e.g., "," or ".").

Handling missing data is essential in data science workflows. ***pandas*** *provides flexible tools for handling* ***NA values****, specifying* ***custom missing value indicators****, and efficiently* ***parsing large files****. By*

leveraging na_values, parse_dates, chunksize, *and other arguments, you can optimize file reading for your dataset.* 🚀

WRITING DATA TO TEXT FORMAT IN PANDAS

pandas provides multiple methods to export **DataFrames** to text-based formats such as **CSV, TSV, JSON, HTML**, and more. The most commonly used function for this is to_csv(), but other functions like to_json(), to_excel(), and to_html() are also available.

1. Writing Data to a CSV File

The to_csv() method writes a **DataFrame** to a **comma-separated values (CSV)** file.

```
import pandas as pd
# Creating a sample DataFrame
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

# Writing to a CSV file
df.to_csv('output.csv', index=False)
```

***Options in** to_csv()*

Table 5-5: Key Arguments for DataFrame.to_csv() Method in Pandas

Argument	Description
path_or_buf	File path or buffer to write to.
Sep	Separator (default is ",", use "\t" for TSV).
Index	Whether to write the row index (default: True).
Header	Whether to include column names (default: True).

Argument	Description
na_rep	String representation of missing values (e.g., "N/A").
Columns	Subset of columns to write.
Mode	File mode ("w" for write, "a" for append).
Encoding	File encoding (e.g., "utf-8").

2. Writing Data to a JSON File

To save a DataFrame in **JSON** format:

```
df.to_json('output.json', orient='records', indent=4)
```

Common **orientations**:

- "records" → List of dictionaries
- "split" → Dictionary with index, columns, data
- "table" → JSON Table Schema

3. Writing Data to an Excel File

To export data to an **Excel (.xlsx)** file:

```
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

4. Writing Data to an HTML File

For exporting tables to **HTML format** (useful for web applications):

```
df.to_html('output.html', index=False)
```

5. Writing Data to a Text File with Fixed Width

For writing data in a **fixed-width format** (without delimiters):

```
df.to_string('output.txt', index=False)
```

QUESTIONS

1. What is the difference between reading (r), writing (w), and appending (a) modes in file handling?
2. How do you open a file in Python and ensure it closes properly?
3. What is the advantage of using the with open() statement over open() and close()?
4. What will happen if you try to write to a file opened in read mode (r)?
5. Write a Python script to create a new text file named data.txt and write the sentence "Python makes file handling easy." into it.
6. How do you read all lines of a file into a list in Python?
7. Explain the difference between read(), readline(), and readlines().
8. Write a Python program to read a file named example.txt and print its contents line by line.
9. How can you write multiple lines into a file using Python?
10. What does the "w" mode do if the file already exists?
11. Write a Python script to read a CSV file named data.csv and print its contents using the csv module. How can you write a pandas DataFrame to a CSV file? Provide an example.
12. Explain the difference between pd.read_csv() and to_csv() in pandas.
13. How do you convert a Python dictionary into a JSON object and save it to a file?
14. Why is it important to handle missing values in data science?

15. How do you check for missing values in a pandas DataFrame?
16. Write a Python script to replace all missing values in a pandas DataFrame with the mean of the respective column.
17. What are the advantages of using Python libraries like pandas over the built-in csv module for handling large datasets?

MODULE 6

DATA STRUCTURES IN PYTHON

Data structures are the shiny gadgets you'll need to store, organize, and juggle information like a true data wizard. Mastering them means you can finally stop treating your data like random piles of socks and start organizing it like a pro.

In this module, readers will:

1. **Understand the Importance of Data Structures** – Learn why data structures are fundamental for efficient data storage and manipulation in Python.
2. **Work with Built-in Data Structures** – Explore lists, tuples, dictionaries, and sets, and understand their properties and use cases.
3. **Manipulate Lists and Tuples** – Learn indexing, slicing, and common operations such as appending, inserting, and removing elements.
4. **Work with Dictionaries and Sets** – Understand key-value pairs, hash-based lookups, and set operations like union, intersection, and difference.
5. **Understand Mutability and Performance** – Learn the difference between mutable and immutable data structures and how they impact performance.
6. **Apply Data Structures in Data Science** – Understand how Python data structures help in data wrangling, preprocessing, and analysis

By the end of this module, readers will be able to choose the right data structure for different programming tasks and efficiently manipulate data in Python.

INTRODUCTION

Data structures are used to store and organize data efficiently. Python provides several built-in data structures, including lists, tuples, sets, and dictionaries.

Think of data structures as the secret filing cabinets of Python. Without them, your data would just be lying around like dirty laundry. Lists, tuples, sets, and dictionaries are not just fancy words; they are the neat little boxes that keep your information sorted, accessible, and ready for action. By truly understanding how they work, you will not just use Python, you will command it like a data scientist who knows exactly where every sock, receipt, and plot twist is stored.

So far we have seen and used these data structures in this book, but still don't understand how it works. This module will provide us with the understanding needed to kick start our journey in data science.

BUILT-IN DATA STRUCTURES, METHODS

This module discusses capabilities built into the Python language that will be used ubiquitously throughout the book. While add-on libraries like pandas and NumPy add advanced computational functionality for larger datasets, they are designed to be used together with Python's built-in data manipulation tools.

1. Lists

Lists are ordered, mutable collections of items. They can contain elements of different data types. Lists in contrast with tuples, are variable-length and their contents can be modified in-place. You can define list using square brackets [] or using the list type function:

Syntax

```
List_name=['Values']
```

```
In [1]: my_list = [2, 3, 7, None]
My_list
```

The list function is frequently used in data processing as a way to materialize an iterator or generator expression:

```
In [2]: gen = range(10)
In [3]: gen
Out[3]: range(0, 10)
In [4]: list(gen)
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Operations performed on List

- **Adding and removing elements**

Elements can be appended to the end of the list with the append method:

```
In [5]: my_list.append('dwarf')
In [6]: my_list
```

Using insert you can insert an element at a specific location in the list:

Syntax:

listName.insert(index, value)

Example:

```
In [7]: b_list.insert(1, 'red')
In [8]: b_list
Out[8]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

The insertion index must be between 0 and the length of the list, inclusive. Insert is computationally expensive compared with append, because references to subsequent elements have to be shifted internally to make room for the new element. If you need to insert elements at both the beginning and end of a sequence, you may wish to explore collections.deque, a double-ended queue, for this purpose.

The inverse operation to insert is pop, which removes and returns an element at a particular index:

Syntax:

listName.pop(index)

```
# Example of a list
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
# Adding an item
fruits.append("orange")
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange']
In [9]: b_list.pop(2)
Out[9]: 'peekaboo'
In [10]: b_list
Out[10]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value with remove, which locates the first such value and removes it from the last:

```
# Removing an item
fruits.remove("banana")
print(fruits)  # Output: ['apple', 'cherry', 'orange']
In [11]: b_list.append('foo')
In [12]: b_list
Out[12]: ['foo', 'red', 'baz', 'dwarf', 'foo']
In [13]: b_list.remove('foo')
In [14]: b_list
Out[14]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using append and remove, you can use a Python list as a perfectly suitable "multiset" data structure.

Check if a list contains a value using the in keyword:

```
In [15]: 'dwarf' in b_list
Out[15]: True
```

The keyword not can be used to negate in:

```
In [16]: 'dwarf' not in b_list
Out[16]: False
```

Checking whether a list contains a value is a lot slower than doing so with dicts and sets (to be introduced shortly), as Python makes

a linear scan across the values of the list, whereas it can check the others (based on hash tables) in constant time.

- **Concatenating and combining lists**

Adding two lists together with + concatenates them:

```
In [17]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[17]: [4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the extend method:

```
In [18]: x = [4, None, 'foo']
In [19]: x.extend([7, 8, (2, 3)])
In [20]: x
Out[20]: [4, None, 'foo', 7, 8, (2, 3)]
```

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using extend to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus,

```
everything = []
for chunk in list_of_lists:
everything.extend(chunk)
is faster than the concatenative alternative:
everything = []
for chunk in list_of_lists:
everything = everything + chunk
```

- **Sorting**

You can sort a list in-place (without creating a new object) by calling its sort function:

```
In [21]: a = [7, 2, 5, 1, 3]
In [22]: a.sort()
In [23]: a
Out[23]: [1, 2, 3, 5, 7]
```

sort has a few options that will occasionally come in handy. One is the ability to pass a secondary *sort key*,that is, a function that

produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [24]: b = ['saw', 'small', 'He', 'foxes', 'six']
In [25]: b.sort(key=len)
In [26]: b
Out[26]: ['He', 'saw', 'six', 'small', 'foxes']
```

Soon, we'll look at the sorted function, which can produce a sorted copy of a general sequence.

- **Binary search and maintaining a sorted list**

The built-in bisect module implements binary search and insertion into a sorted list. bisect.bisect finds the location where an element should be inserted to keep it sorted, while bisect.insort actually inserts the element into that location:

```
In [27]: import bisect
In [28]: c = [1, 2, 2, 2, 3, 4, 7]
In [29]: bisect.bisect(c, 2)
Out[29]: 4
In [30]: bisect.bisect(c, 5)
Out[30]: 6
In [31]: bisect.insort(c, 6)
In [32]: c
Out[32]: [1, 2, 2, 2, 3, 4, 6, 7]
```

The bisect module functions do not check whether the list is sorted, as doing so would be computationally expensive. Thus, using them with an unsorted list will succeed without error but may lead to incorrect results.

Slicing in Lists in Python

List slicing allows you to access a subset of a list using the **slice notation**:

list[start:stop:step]

- **start** – Index where the slice begins (inclusive).
- **stop** – Index where the slice ends (exclusive).

- **step** – Specifies the step between indices (default is 1).

Basic List Slicing

```
numbers = [10, 20, 30, 40, 50, 60, 70]
print(numbers[1:4])  # Output: [20, 30, 40] (Indexes 1 to 3)
print(numbers[:3])   # Output: [10, 20, 30] (Start from beginning)
print(numbers[3:])   # Output: [40, 50, 60, 70] (Till the end)
print(numbers[:])    # Output: [10, 20, 30, 40, 50, 60, 70] (Entire list)
```

Using Negative Indices

```
print(numbers[-3:])  # Output: [50, 60, 70] (Last 3 elements)
print(numbers[:-2])  # Output: [10, 20, 30, 40, 50] (Exclude last 2)
print(numbers[-5:-2])  # Output: [30, 40, 50]
```

Using Step Values

```
print(numbers[::2])   # Output: [10, 30, 50, 70] (Every second element)
print(numbers[::-1])  # Output: [70, 60, 50, 40, 30, 20, 10] (Reverse list)
print(numbers[1:6:2]) # Output: [20, 40, 60] (From index 1 to 5, step 2)
```

Modifying Lists with Slicing

Slicing can be used to **modify** lists by replacing elements.

```
numbers[2:5] = [300, 400]  # Replace elements at index 2,3,4
print(numbers)  # Output: [10, 20, 300, 400, 60, 70]
```

Deleting Elements using Slicing

```
numbers[2:4] = []  # Remove elements at index 2 and 3
print(numbers)  # Output: [10, 20, 60, 70]
del numbers[1:3]  # Delete elements at index 1 and 2
print(numbers)  # Output: [10, 70]
```

List slicing is a powerful technique in Python to extract, modify, or delete elements efficiently. It provides flexibility with start, stop, and step parameters, making it useful for data manipulation and sequence processing.

- **Reversed**

Reversed iterates over the elements of a sequence in reverse order:

```
In [40]: list(reversed(range(10)))
Out[40]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Keep in mind that reversed is a generator, so it does not create the reversed sequence until materialized (e.g., with list or a for loop).

Table 6-1: Common Python List Operations: Methods, Descriptions, and Examples

Operation/Method	Description	Example	Output
list[index]	Accesses the element at the given index	lst = [1, 2, 3]; lst[1]	2
list[index] = value	Modifies the element at the given index	lst[1] = 5	[1, 5, 3]
list.append(value)	Adds an element to the end of the list	lst.append(4)	[1, 2, 3, 4]
list.extend(iterable)	Appends elements from an iterable	lst.extend([5, 6])	[1, 2, 3, 5, 6]
list.insert(index, value)	Inserts an element at the specified index	lst.insert(1, 10)	[1, 10, 2, 3]
list.remove(value)	Removes the first occurrence of value	lst.remove(2)	[1, 3]
list.pop(index)	Removes and returns the element at index (last if omitted)	lst.pop(1)	2
list.clear()	Removes all elements from the list	lst.clear()	[]
list.index(value, start, end)	Returns the index of value within optional range	lst.index(3)	2
list.count(value)	Counts occurrences of value in the list	lst.count(2)	1

Operation/Method	Description	Example	Output
list.sort(reverse=False, key=func)	Sorts the list in place	lst.sort()	[1, 2, 3]
list.reverse()	Reverses the list in place	lst.reverse()	[3, 2, 1]
sorted(list, reverse=False, key=func)	Returns a sorted copy of the list	sorted(lst)	[1, 2, 3]
list.copy()	Returns a shallow copy of the list	lst.copy()	[1, 2, 3]
list * n	Repeats the list n times	[1, 2] * 2	[1, 2, 1, 2]
list + list2	Concatenates two lists	[1, 2] + [3, 4]	[1, 2, 3, 4]
del list[index]	Deletes the element at index	del lst[1]	[1, 3]
del list[start:end]	Deletes a slice of elements	del lst[1:3]	[1]
len(list)	Returns the length of the list	len(lst)	3
max(list)	Returns the maximum element	max([1, 2, 3])	3
min(list)	Returns the minimum element	min([1, 2, 3])	1
sum(list)	Returns the sum of elements	sum([1, 2, 3])	6
value in list	Checks if value exists in the list	3 in lst	True

Operation/Method	Description	Example	Output
value not in list	Checks if value is absent	4 not in lst	True

2. Tuples

Tuples in Python are similar to lists, except for their key feature of being immutable. Once a tuple has been created, it will remain the same throughout its lifespan. Elements cannot be added to or deleted from a tuple once it has been created which makes it a suitable choice for storing data that does not change over time. They are often initialized by listing their contents between parentheses or brackets.

```
In [1]: tup = 4, 5, 6
In [2]: tup
Out[2]: (4, 5, 6)
```

Sometimes you need parentheses around values within a tuple when constructing more complex expressions such as when forming a nested tuple, e.g :

```
In [3]: nested_tup = (4, 5, 6), (7, 8)
In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

Converting any sequence or iterator into a tuple is achieved by using the tuple function, e.g :

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)
In [6]: tup = tuple('string')
In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

You can use square brackets to fetch items from the sequence, like with many other object types. As in C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [8]: tup[0]
Out[8]: 's'
```

While the objects stored in a tuple may be mutable themselves, once the tuple is created it's not possible to modify which object is stored in each slot:

```
In [9]: tup = tuple(['foo', [1, 2], True])
In [10]: tup[2] = False
```

TypeError Traceback (most recent call last)
<ipython-input-10-c7308343b841> **in** <module>()
---> 1 tup[2] = False
TypeError: 'tuple' object does **not** support item assignment

If an object inside a tuple is mutable, such as a list, you can modify it in-place:

```
In [11]: tup[1].append(3)
In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

You can concatenate tuples using the + operator to produce longer tuples:

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Note that the objects themselves are not copied, only the references to them. If you try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* the value on the righthand side of the equals sign:

```
In [15]: tup = (4, 5, 6)
In [16]: a, b, c = tup
In [17]: b
Out[17]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [18]: tup = 4, 5, (6, 7)
In [19]: a, b, (c, d) = tup
In [20]: d
Out[20]: 7
```

Using this functionality you can easily swap variable names, a task which in many languages might look like:

```
tmp = a
a = b
b = tmp
```

But, in Python, the swap can be done like this:

```
In [21]: a, b = 1, 2
In [22]: a
Out[22]: 1
In [23]: b
Out[23]: 2
In [24]: b, a = a, b
In [25]: a
Out[25]: 2
In [26]: b
Out[26]: 1
```

A common use of variable unpacking is iterating over sequences of tuples or lists:

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [28]: for a, b, c in seq:
....: print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

The Python language recently acquired some more advanced tuple unpacking to help with situations where you may want to “pluck” a few elements from the beginning of a tuple. This uses the special syntax *rest, which is also used in function signatures to capture an arbitrarily long list of positional arguments:

```
In [29]: values = 1, 2, 3, 4, 5
```

```
In [30]: a, b, *rest = values
In [31]: a, b
Out[31]: (1, 2)
In [32]: rest
Out[32]: [3, 4, 5]
```

This rest bit is sometimes something you want to discard; there is nothing special about the rest name. As a matter of convention, many Python programmers will use the underscore (_) for unwanted variables:

```
In [33]: a, b, *_ = values
```

Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. A particularly useful one (also available on lists) is count, which counts the number of occurrences of a value:

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)
In [35]: a.count(2)
Out[35]: 4
```

Table 6-2: Essential Tuple Methods and Operations in Python

Operation/Method	Description	Example	Output
tuple[index]	Accesses the element at the given index	t = (1, 2, 3); t[1]	2
tuple[start:end]	Returns a slice from start to end-1	t[1:3]	(2, 3)
len(tuple)	Returns the number of elements in the tuple	len(t)	3
tuple.count(value)	Counts occurrences	t.count(2)	1

Operation/Method	Description	Example	Output
	of value in the tuple		
tuple.index(value, start, end)	Returns the index of value in the optional range	t.index(3)	2
tuple1 + tuple2	Concatenates two tuples	(1, 2) + (3, 4)	(1, 2, 3, 4)
tuple * n	Repeats the tuple n times	(1, 2) * 2	(1, 2, 1, 2)
value in tuple	Checks if value exists in the tuple	3 in t	True
value not in tuple	Checks if value is absent	4 not in t	True
min(tuple)	Returns the smallest element	min((3, 1, 2))	1
max(tuple)	Returns the largest element	max((3, 1, 2))	3
sum(tuple)	Returns the sum of elements	sum((1, 2, 3))	6
tuple(sorted(iterable))	Returns a sorted version of an iterable as a tuple	tuple(sorted((3, 1, 2)))	(1, 2, 3)
tuple(iterable)	Converts an iterable (list, set, etc.) into a tuple	tuple([1, 2, 3])	(1, 2, 3)
del tuple	Deletes the entire tuple (not an element)	del t	*Tuple is deleted*

Note:

- **Tuples are immutable**, so operations like tuple.append(), tuple.remove(), and tuple.sort() are **not allowed**.

```
# Tuples are immutable
coordinates = (10, 20)
# coordinates[0] = 15  # This will raise an error
```

- If modification is required, convert the tuple to a list:

```
coordinates = (10, 20)
lst = list(coordinates)
lst.append(30)
my_new_lst = tuple(lst)
print(my_new_lst)  # Output: (10, 20, 30)
```

3. Sets

Sets are unordered collections of unique elements. A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the set function or via a *set literal* with curly braces:

```
#Creating a set via the set literal
In [1]: set([2, 2, 2, 1, 3, 3])
Out[2]: {1, 2, 3}

#Creating a set with curly braces
In [3]: {2, 2, 2, 1, 3, 3}
Out[4]: {1, 2, 3}
```

Like the mathematical set, The Set data structure supports mathematical *set operations* like union, intersection, difference, and symmetric difference. Consider these two example sets:

```
In [5]: a = {1, 2, 3, 4, 5}
In [6]: b = {3, 4, 5, 6, 7, 8}
```

The union of these two sets is the set of distinct elements occurring in either set. This can be computed with either the union method or the | binary operator:

```
In [7]: a.union(b)
Out[7]: {1, 2, 3, 4, 5, 6, 7, 8}
In [8]: a | b
Out[8]: {1, 2, 3, 4, 5, 6, 7, 8}
```

The intersection contains the elements occurring in both sets. The & operator or the intersection method can be used:

```
In [9]: a.intersection(b)
Out[9]: {3, 4, 5}
In [10]: a & b
Out[10]: {3, 4, 5}
```

Table 6-3: Set Methods

Method/Operator	Description	Example	Output
set.add(element)	Adds an element to the set. If the element already exists, it is ignored.	s = {1, 2}; s.add(3); print(s)	{1, 2, 3}
set.remove(element)	Removes the specified element. Raises KeyError if the element is not found.	s = {1, 2, 3}; s.remove(2); print(s)	{1, 3}
set.discard(element)	Removes the specified element if it exists. No error if the element is not found.	s = {1, 2, 3}; s.discard(4); print(s)	{1, 2, 3}
set.pop()	Removes and returns an arbitrary	s = {1, 2, 3}; print(s.pop())	1 (or any other element)

	element from the set. Raises KeyError if empty.		
set.clear()	Removes all elements from the set.	s = {1, 2, 3}; s.clear(); print(s)	set()
set.copy()	Returns a shallow copy of the set.	s1 = {1, 2, 3}; s2 = s1.copy(); print(s2)	{1, 2, 3}
set.union(*others) or `A B`		Returns a new set containing all elements from both sets.	`{1, 2}
set.intersection(*others) or A & B	Returns a new set containing only common elements.	{1, 2} & {2, 3}	{2}
set.difference(*others) or A – B	Returns a new set with elements in A but	{1, 2, 3} - {2, 3}	{1}

	not in B.		
set.symmetric_difference(other) or A ^ B	Returns a set with elements in either A or B but not both.	{1, 2, 3} ^ {2, 3, 4}	{1, 4}
set.update(*others) or `A	= B`	Updates the set by adding elements from another set.	s = {1, 2}; s.update({2, 3, 4}); print(s)
set.intersection_update(*others) or A &= B	Updates the set, keeping only elements found in both.	s = {1, 2, 3}; s.intersection_update({2, 3, 4}); print(s)	{2, 3}
set.difference_update(*others) or A -= B	Updates the set, removing elements found in B.	s = {1, 2, 3}; s.difference_update({2, 3, 4}); print(s)	

Like dicts, set elements generally must be immutable. To have list-like elements, you must convert it to a tuple:

```
In [17]: my_data = [1, 2, 3, 4]
```

```
In [18]: my_set = {tuple(my_data)}
In [19]: my_set
Out[19]: {(1, 2, 3, 4)}
```

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [20]: a_set = {1, 2, 3, 4, 5}
In [21]: {1, 2, 3}.issubset(a_set)
Out[21]: True
In [22]: a_set.issuperset({1, 2, 3})
Out[22]: True
```

Sets are equal if and only if their contents are equal:

```
In [23]: {1, 2, 3} == {3, 2, 1}
Out[23]: True
```

4. Dictionaries

Dictionaries store data as key-value pairs. Keys must be unique and immutable. dict is likely the most important built-in Python data structure. A more common name for it is *hash map* or *associative array*. It is a flexibly sized collection of *key-value* pairs, where *key* and *value* are Python objects. One approach for creating one is to use curly braces {} and colons to separate keys and values:

```
# Example of a dictionary
person = {"name": "Alice", "age": 25}
print(person["name"])  # Output: Alice
In [1]: empty_dict = {}
In [2]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
In [3]: d1
Out[3]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

You can access, insert, or set elements using the same syntax as for accessing elements
of a list or tuple:

```
# Adding a new key-value pair
person["city"] = "New York"
print(person)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
In [4]: d1[7] = 'an integer'
In [5]: d1
Out[5]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
In [6]: d1['b']
Out[6]: [1, 2, 3, 4]
```

You can check if a dict contains a key using the same syntax used for checking whether a list or tuple contains a value:

```
In [7]: 'b' in d1
Out[7]: True
```

You can delete values either using the del keyword or the pop method (which simultaneously returns the value and deletes the key):

```
In [8]: d1[5] = 'some value'
In [9]: d1
Out[9]:
{'a': 'some value',
'b': [1, 2, 3, 4],
7: 'an integer',
5: 'some value'}
In [10]: d1['dummy'] = 'another value'
In [11]: d1
Out[11]:
{'a': 'some value',
'b': [1, 2, 3, 4],
7: 'an integer',
5: 'some value',
'dummy': 'another value'}
In [12]: del d1[5]
In [13]: d1
Out[13]:
{'a': 'some value',
'b': [1, 2, 3, 4],
7: 'an integer',
'dummy': 'another value'}
```

```
In [14]: ret = d1.pop('dummy')
In [15]: ret
Out[15]: 'another value'
In [16]: d1
Out[16]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

The keys and values method give you iterators of the dict's keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

```
In [17]: list(d1.keys())
Out[17]: ['a', 'b', 7]
In [18]: list(d1.values())
Out[18]: ['some value', [1, 2, 3, 4], 'an integer']
```

You can merge one dict into another using the update method:

```
In [19]: d1.update({'b' : 'foo', 'c' : 12})
In [20]: d1
Out[20]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

The update method changes dicts in-place, so any existing keys in the data passed to update will have their old values discarded.

Table 6-4: Dict Operations

Operation	Description	Example	Output
dict[key]	Accesses the value associated with key	d = {"a": 1}; d["a"]	1
dict[key] = value	Adds or updates a key-value pair	d["b"] = 2	{"a": 1, "b": 2}
del dict[key]	Deletes the key-value pair	del d["a"]	{"b": 2}
dict.get(key, default)	Retrieves the	d.get("c", 0)	0

Operation	Description	Example	Output
	value for key, or default if not found		
dict.keys()	Returns a view object of dictionary keys	d.keys()	dict_keys(["a", "b"])
dict.values()	Returns a view object of dictionary values	d.values()	dict_values([1, 2])
dict.items()	Returns a view object of key-value pairs	d.items()	dict_items([("a", 1), ("b", 2)])
dict.pop(key, default)	Removes and returns the value for key, or default if not found	d.pop("a", 0)	1
dict.popitem()	Removes and returns the last key-value pair (LIFO order in Python 3.7+)	d.popitem()	("b", 2)
dict.clear()	Removes all items from the dictionary	d.clear()	{}

Operation	Description	Example	Output
dict.copy()	Returns a shallow copy of the dictionary	d.copy()	{"a": 1, "b": 2}
dict.update(other_dict)	Merges another dictionary into dict	d.update({"c": 3})	{"a": 1, "b": 2, "c": 3}
dict.setdefault(key, default)	Returns the value of key, or sets it to default if key is missing	d.setdefault("d", 4)	4
dict.fromkeys(iterable, value)	Creates a new dictionary from keys with a default value	dict.fromkeys(["x", "y"], 0)	{"x": 0, "y": 0}
{**dict1, **dict2}	Merges two dictionaries (Python 3.5+)	{**{"a": 1}, **{"b": 2}}	{"a": 1, "b": 2}
`dict	other_dict`	Merges dictionaries (Python 3.9+)	`{"a": 1}
`dict	= other_dict`	In-place merge update (Python 3.9+)	`d
key in dict	Checks if key	"a" in d	True

Operation	Description	Example	Output
	exists in the dictionary		
len(dict)	Returns the number of key-value pairs	len(d)	2

LIST, DICT, SET COMPREHENSIONS

List comprehensions provide a concise way to create lists. **List, Set, and Dict Comprehensions**

List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression. They take the basic form:

[*expr* for val in collection if *condition*]

This is equivalent to the following for loop:

```
result = []
for val in collection:
if condition:
result.append(expr)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

```
In [24]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
In [25]: [x.upper() for x in strings if len(x) > 2]
Out[25]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
# Example of a list comprehension
squares = [x ** 2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Set and dict comprehensions are a natural extension, producing sets and dicts in an idiomatically similar way instead of lists. A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection
if condition}
```

A set comprehension looks like the equivalent list comprehension except with curly
braces instead of square brackets:

```
set_comp = {expr for value in collection if condition}
```

Like list comprehensions, set and dict comprehensions are mostly conveniences, but they similarly can make code both easier to write and read. Consider the list of strings from before. Suppose we wanted a set containing just the lengths of the strings contained in the collection; we could easily compute this using a set comprehension:

```
In [26]: unique_lengths = {len(x) for x in strings}
In [27]: unique_lengths
Out[27]: {1, 2, 3, 4, 6}
```

We could also express this more functionally using the map function, introduced shortly:

```
In [28]: set(map(len, strings))
Out[28]: {1, 2, 3, 4, 6}
```

As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list:

```
In [29]: loc_mapping = {val : index for index, val in enumerate(strings)}
In [30]: loc_mapping
Out[30]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Nested list comprehensions

Suppose we have a list of lists containing some English and Spanish names:

```
In [31]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
.....: ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

You might have gotten these names from a couple of files and decided to organize them by language. Now, suppose we wanted to get a single list containing all names with two or more e's in them. We could certainly do this with a simple for loop:

```
names_of_interest = []
for names in all_data:
enough_es = [name for name in names if name.count('e') > = 2]
names_of_interest.extend(enough_es)
```

You can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [32]: result = [name for names in all_data for name in names
.....: if name.count('e') > = 2]
In [33]: result
Out[33]: ['Steven']
```

At first, nested list comprehensions are a bit hard to wrap your head around. The for parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before. Here is another example where we "flatten" a list of tuples of integers into a simple list of integers:

```
In [34]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [35]: flattened = [x for tup in some_tuples for x in tup]
In [36]: flattened
Out[36]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that the order of the for expressions would be the same if you wrote a nested for loop instead of a list comprehension:

```
flattened = []
for tup in some_tuples:
for x in tup:
flattened.append(x)
```

You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question whether this makes sense from a code readability standpoint. It's important to distinguish the syntax just

shown from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [37]: [[x for x in tup] for tup in some_tuples]
Out[37]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

This produces a list of lists, rather than a flattened list of all of the inner elements.

APPLICATION OF DATA STRUCTURES IN DATA SCIENCE

In data science, choosing the right data structure is crucial for efficient data storage, retrieval, and manipulation. We will cover the four fundamental data structures,**Lists, Tuples, Dictionaries, and Sets**,in detail, including their applications, examples, and Python implementations.

1. **Lists**

A **list** is a mutable, ordered collection of elements that can store different data types. It allows easy addition, deletion, and modification of elements.

Applications in Data Science

a. Storing datasets before converting them into structured formats (e.g., Pandas DataFrames).
b. Holding multiple values retrieved from a dataset.
c. Iterating over sequences in data processing.

Example

Imagine we have a dataset of sales revenue recorded daily. We can store it in a list and perform calculations.

```
# List of daily sales revenue
daily_sales = [2500, 3000, 2700, 4000, 3200, 2900, 3100]
```

```
# Calculate total revenue
total_revenue = sum(daily_sales)
print("Total Revenue:", total_revenue)

# Finding the maximum and minimum sales in a week
print("Highest Sales:", max(daily_sales))
print("Lowest Sales:", min(daily_sales))
```

Output

```
Total Revenue: 21400
Highest Sales: 4000
Lowest Sales: 2500
```

2. **Tuples**

A **tuple** is an immutable, ordered collection of elements. It is used when data should remain unchanged throughout execution.

Applications in Data Science

a. Storing fixed metadata (e.g., coordinates, feature names).
b. Using tuples as keys in dictionaries (since they are immutable).
c. Faster access to data compared to lists due to immutability.

Example

Let's consider a dataset where each data point represents a geographical location (latitude, longitude).

```
# Tuple representing a location (Latitude, Longitude)
location = (37.7749, -122.4194)  # San Francisco

# Accessing values
print("Latitude:", location[0])
print("Longitude:", location[1])

# Tuples as dictionary keys (efficient for lookups)
temperature_data = {
   (37.7749, -122.4194): 15.5,  # Temperature in Celsius
   (40.7128, -74.0060): 10.2    # New York
```

```
}
print("Temperature in San Francisco:", temperature_data[location])
```

Output

```
Latitude: 37.7749
Longitude: -122.4194
Temperature in San Francisco: 15.5
```

3. Dictionaries

A **dictionary** is a collection of key-value pairs that provides fast lookups and is widely used in data science.

Applications in Data Science

a. Mapping categorical variables to numerical values.
b. Storing data in key-value format for fast lookups.
c. Aggregating statistics from raw data.

Example

Suppose we are working with customer purchase records where each customer has a unique ID.

```
# Dictionary of customers and their total purchase amount
customer_purchases = {
   "C001": 1200.50,
   "C002": 850.75,
   "C003": 430.60,
   "C004": 1540.90
}

# Accessing purchase amount for a specific customer
print("Customer C002's purchase amount:", customer_purchases["C002"])

# Adding a new customer record
customer_purchases["C005"] = 920.30

# Iterating through dictionary
for customer, amount in customer_purchases.items():
```

```
print(f"Customer {customer} spent ${amount}")
```

Output

```
Customer C002's purchase amount: 850.75
Customer C001 spent $1200.5
Customer C002 spent $850.75
Customer C003 spent $430.6
Customer C004 spent $1540.9
Customer C005 spent $920.3
```

4. **Sets**

A **set** is an unordered collection of unique elements. It is useful for eliminating duplicate values.

Applications in Data Science

a. Removing duplicate entries from a dataset.
b. Performing set operations (union, intersection) on categorical data.
c. Checking for membership in large datasets.

Example

Let's say we have a dataset containing email IDs of registered users, but there are duplicates.

```
# List of email IDs (some are repeated)
email_list = ["user1@example.com", "user2@example.com", "user3@example.com",
        "user1@example.com", "user4@example.com", "user2@example.com"]

# Convert list to set to remove duplicates
unique_emails = set(email_list)

print("Unique Email IDs:", unique_emails)
```

Output

```
Unique Email IDs: {'user1@example.com', 'user3@example.com', 'user4@example.com', 'user2@example.com'}
```

QUESTIONS

1. What is the difference between a list and a tuple?
2. Create a list of integers from 1 to 10. Write code to reverse the list and remove the last element.
3. Given a tuple t = (5, 10, 15, 20), write code to access the second element and print it.
4. Create a list of numbers and use a list comprehension to filter out even numbers.
5. Given a list
6. Write a Python program to count the frequency of each element in a list using a dictionary.
7. What is the advantage of using sets over lists?
8. Create a dictionary to store student names and their corresponding grades. Add a new student and update an existing student's grade.
9. How do data structures contribute to efficient data manipulation in data science?
10. Given two sets, A = {1, 2, 3, 4} and B = {3, 4, 5, 6}, write code to find their intersection and union.
11. Write a Python program to append the number 7 to a list [1, 2, 3, 4, 5] and then insert the number 6 at the second position.
12. Create a dictionary where the keys are student names and the values are their grades. Write code to check if a specific student exists in the dictionary.
13. Given a list [10, 20, 30, 40, 50], write code to slice the list and extract the elements [20, 30, 40].
14. Write a Python program to remove duplicates from a list [1, 2, 2, 3, 4, 4, 5] using a set.

15. Create a tuple with mixed data types (e.g., integers, strings, floats). Write code to print its length and the data type of the third element.
16. Given a dictionary {'a': 1, 'b': 2, 'c': 3}, write code to iterate through its keys and values,

MODULE 7

DATA MANIPULATION AND ANALYSIS WITH NUMPY AND PANDAS

Data manipulation and Analysis is where raw, messy data transforms into polished, meaningful insights. In this module, we roll up our sleeves and dive into the real action: **manipulating** and **analyzing** data like pros. Armed with **NumPy** and **Pandas**, two of Python's most powerful libraries, you will learn how to bend data to your will. NumPy will be your go-to toolkit for high-speed numerical computations, while Pandas will become your best friend for working with structured data.
By the end of this module, the reader should be able to:

Work with the NumPy and Pandas Libraries

- Understand NumPy arrays and their advantages over Python lists.
- Create and manipulate NumPy arrays (1D, 2D, and multi-dimensional).
- Perform element-wise operations and broadcasting.
- Use NumPy for mathematical and statistical operations.
- Work with indexing, slicing, and reshaping arrays.
- Handle missing values and NaN in NumPy.

Carry out Data Handling with Pandas Library

- Understand the Pandas Series and DataFrame structures.
- Create and modify Pandas DataFrames from lists, dictionaries, and CSV/Excel files.

- Perform indexing, selection, and filtering on DataFrames.
- Handle missing data (detection, filling, and dropping).

Carry out Data Manipulation with Pandas

- Perform data cleaning (handling duplicates, missing values, and data types).
- Use Pandas functions for data transformation (apply, map, groupby, pivot tables).
- Perform merging, joining, and concatenation of datasets.
- Work with time series data using Pandas.

Perform Data Analysis with NumPy and Pandas

- Perform descriptive statistics (mean, median, mode, standard deviation).
- Aggregate and summarize data using Pandas.
- Use NumPy for linear algebra and matrix operations.
- Perform correlation and covariance analysis with Pandas.

Carryout BASIC Visualization with Pandas

- Generate basic plots using Pandas' built-in visualization.
- Integrate Pandas with Matplotlib and Seaborn for advanced visualizations.

Apply both libraries in Real-world scenarios

- Apply NumPy and Pandas to real-world datasets (e.g., stock market, weather, customer transactions).
- Optimize performance in large datasets using Pandas' vectorized operations.

Introduction

Data manipulation is a crucial step in data science, allowing researchers and analysts to clean, transform, and prepare data for analysis. Python provides powerful libraries for data manipulation, with NumPy and Pandas being the most widely used.

In the world of data science, data manipulation is like preparing jollof rice. If you do not clean, sort, and cook the ingredients properly, you will serve nonsense and nobody will consume the meal. Raw data is usually messy, scattered, and stubborn like Lagos traffic on a Monday morning. That is why Python gave us NumPy and Pandas, the real MVPs, to help us arrange, transform, and polish our data until it is sweet and ready for analysis. With these tools, you will not just manage data, you will package it like a correct boss!

Why NumPy and Pandas?

- **NumPy (Numerical Python):** Efficiently handles numerical data, providing powerful multi-dimensional array operations and mathematical functions.
- **Pandas:** Designed for data manipulation and analysis, offering easy-to-use data structures like Series and DataFrames.

Installing and Importing NumPy and Pandas

Jupyter Notebook comes with lots of preinstalled libraries (including NumPy and Pandas). To install NumPy and Pandas if they are not pre-installed on your computer, input the following command:

```
pip install numpy pandas
```

Note: Ensure your device is connected to a network to enable download of this library.

Importing the libraries in Python:

```
import numpy as np
import pandas as pd
```

WORKING WITH NUMPY

NumPy (Numerical Python) is a fundamental library for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions.

Understanding NumPy Arrays

NumPy arrays are faster than Python lists because they enable more powerful operations on arrays.

Arrays and Matrix Operations

Creating NumPy Arrays

NumPy arrays behave like Python lists, but are optimized for performing numerical operations.

```
# Creating a 1D array
import numpy as np

# Creating a 1D array
arr = np.array([1, 2, 3, 4, 5])
print(arr)  # Output: [1 2 3 4 5]

arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)



# Creating a 2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
# Creating a 2D array
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix)
# Output:
```

```
# [[1 2 3]
#  [4 5 6]]
```

Array Attributes

NumPy arrays include attribute properties such as shape, type and size.

```
print(arr.shape)    # Output: (5,) (1D array with 5 elements)
print(matrix.shape) # Output: (2, 3) (2 rows, 3 columns)
print(arr.dtype)    # Output: int64 (data type of elements)
print(arr.size)     # Output: 5 (total number of elements)
```

Matrix Operations

NumPy allows performing multiple matrix operations, including addition, multiplication and dot products.

```
# Matrix addition
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(a + b)
```

```
# Output:
# [[ 6  8]
#  [10 12]]
```

```
# Matrix multiplication
print(np.dot(a, b))
```

```
# Output:
# [[19 22]
#  [43 50]]
```

Element-wise Operations and Broadcasting

NumPy allows performing mathematical calculations on arrays.

```
arr = np.array([10, 20, 30])
print(arr + 5)  # Broadcasting: Adds 5 to each element
```

Broadcasting permits NumPy to work with arrays in a wide range of sizes and structures.

```
# Example of broadcasting
```

```
a = np.array([1, 2, 3])
b = 2
```

```
print(a * b)  # Output: [2 4 6]

# Broadcasting with 2D arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([10, 20])
print(a + b)
```

```
# Output:
# [[11 22]
#  [13 24]]
```

Mathematical and Statistical Operations

```
arr = np.array([10, 20, 30, 40])
print("Mean:", np.mean(arr))
print("Sum:", np.sum(arr))
print("Standard Deviation:", np.std(arr))
```

Indexing, Slicing, and Reshaping

```
arr = np.array([1, 2, 3, 4, 5, 6])
print(arr[1:4])  # Slicing
arr2d = arr.reshape(2, 3)  # Reshaping
print(arr2d)
```

Handling Missing Values in NumPy

```
arr = np.array([1, 2, np.nan, 4])
print("Mean ignoring NaN:", np.nanmean(arr))
```

DATA HANDLING WITH PANDAS

Introduction to Pandas Series and DataFrame

Pandas is designed to simplify numerous tasks involved in dealing with and analyzing data. DataFrames and Series make it simple to handle and manage structured data.

Series

A Series is similar to Python's lists but offers many useful attributes and options.

Creating a Pandas Series

```
import pandas as pd
# Creating a Series
s = pd.Series([1, 2, 3, 4, 5])
print(s)
```

```
# Output:
# 0  1
# 1  2
# 2  3
# 3  4
# 4  5
# dtype: int64
```

Example 2

```
s = pd.Series([10, 20, 30, 40], index=["A", "B", "C", "D"])
print(s)
```

DataFrames

A DataFrame has rows and columns which resemble a table.

Creating a Pandas DataFrame

```
# Creating a DataFrame
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print(df)
```

```
# Output:
#      Name  Age         City
# 0   Alice   25     New York
# 1     Bob   30  Los Angeles
# 2 Charlie   35      Chicago
```

Example 2

```
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data)
```

```
print(df)
```

Indexing, Selection, and Filtering

```
print(df["Name"])  # Selecting a column
print(df[df["Age"] > 25])  # Filtering rows
```

Handling Missing Data

Many datasets found in the real world contain missing values. Methods are available within Pandas to address problems related to missing values.

```
# Example of missing values
df = pd.DataFrame({
   'A': [1, 2, None, 4],
   'B': [5, None, None, 8],
   'C': [10, 11, 12, 13]
})

# Fill missing values with 0
df_filled = df.fillna(0)
print(df_filled)
```

```
# Output:
#     A    B   C
# 0  1.0  5.0  10
# 1  2.0  0.0  11
# 2  0.0  0.0  12
# 3  4.0  8.0  13
```

```
# Drop rows with missing values
df_dropped = df.dropna()
print(df_dropped)
# Output:
#     A    B   C
# 0  1.0  5.0  10
# 3  4.0  8.0  13
```

Example :

```
df.loc[3] = ["David", np.nan]  # Adding a missing value
print(df.dropna())  # Dropping missing values
print(df.fillna(30))  # Filling missing values with 30
```

Data Transformation

You can transform data by changing data types, applying functions, or creating new columns.

```
# Changing data types
df['A'] = df['A'].astype(float)

# Applying functions
df['B'] = df['B'].apply(lambda x: x * 2 if pd.notnull(x) else x)

# Creating new columns
df['D'] = df['A'] + df['B']
print(df)
```

```
# Output:
#     A     B   C     D
# 0  1.0  10.0  10  11.0
# 1  2.0   NaN  11   NaN
# 2  NaN   NaN  12   NaN
# 3  4.0  16.0  13  20.0
```

DATA MANIPULATION WITH PANDAS

Merging DataFrames

Merging combines DataFrames based on a common column.

```
df1 = pd.DataFrame({
   'Key': ['A', 'B', 'C'],
   'Value': [1, 2, 3]
})

df2 = pd.DataFrame({
   'Key': ['A', 'B', 'D'],
   'Value': [4, 5, 6]
})

# Inner join
merged_df = pd.merge(df1, df2, on='Key', how='inner')
print(merged_df)
```

```
# Output:
#   Key  Value_x  Value_y
```

```
# 0  A      1      4
# 1  B      2      5
```

Concatenating DataFrames

Concatenation combines DataFrames along a specified axis.

```
concatenated_df = pd.concat([df1, df2], axis=0)
print(concatenated_df)
```

```
# Output:
#   Key  Value
# 0  A     1
# 1  B     2
# 2  C     3
# 0  A     4
# 1  B     5
# 2  D     6
```

Cleaning Data

```
df["Age"] = df["Age"].astype(int)  # Converting data type
print(df.drop_duplicates())  # Removing duplicates
```

Applying Functions

```
def age_category(age):
    return "Young" if age < 30 else "Old"
df["Category"] = df["Age"].apply(age_category)
print(df)
```

DATA ANALYSIS WITH NUMPY AND PANDAS

Descriptive Statistics

```
print(df.describe())  # Summary statistics
```

Aggregation and Summarization

```
grouped = df.groupby("Category")["Age"].mean()
print(grouped)
```

Correlation and Covariance Analysis

```
print(df.corr())  # Correlation matrix
```

BASIC VISUALIZATION WITH PANDAS

Basic Pandas Plotting

```
df.plot(kind='bar', x='Name', y='Age')
```

Integrating with Matplotlib and Seaborn

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.histplot(df['Age'])
plt.show()
```

REAL-WORLD APPLICATIONS

Case Study 1: Stock Market Data Analysis

```
df = pd.read_csv("stock_data.csv")
print(df.head())  # Exploring stock data
```

Case Study 2: Weather Data Processing

```
df = pd.read_csv("weather_data.csv")
print(df.groupby("City")["Temperature"].mean())
```

Case Study 3: Customer Transactions Analysis

```
df = pd.read_csv("transactions.csv")
print(df.groupby("CustomerID")["Amount"].sum())
```

Optimizing Performance

```
df["NewColumn"] = df["Age"].apply(lambda x: x * 2)  # Vectorized operations
```

QUESTIONS

1. How does broadcasting work in NumPy?
2. Explain the difference between Pandas Series and DataFrame.
3. What function is used to detect missing values in Pandas?
4. How do you merge two DataFrames in Pandas?
5. What is the purpose of the groupby function in Pandas?
6. Create a 3x3 NumPy array and print its shape and data type.
7. Perform matrix multiplication on two 2x2 arrays.
8. Use broadcasting to add a scalar value to a 1D array.
9. What is the difference between np.dot() and np.multiply()?
10. Create a 2D array and calculate the sum of each row.
11. Create a DataFrame and fill missing values with the mean of the column.
12. Write a Python script to merge two DataFrames based on a common column.
13. Use a lambda function to transform a column in a DataFrame.
14. What is the difference between pd.merge() and pd.concat()?
15. Create a DataFrame and calculate the sum of each column.
16. Write a Pandas script to load a CSV file and display the first five rows.
17. Generate a DataFrame with random numbers and compute its mean and standard deviation.
18. Create a visualization using Pandas' built-in plotting functions for table 7-1

    Table 7-1 shows the sales performance of different products for a specific month, measured in units.

- ***X-axis****: Represents the products (Product A, Product B, etc.).*
- ***Y-axis****: Represents the sales in units.*
- *Each bar corresponds to the sales performance of a specific product.*

Product	**Sales(in units)**
Product A	120
Product B	85
Product C	150.82
Product D	60.5
Product E	200
Product F	70.4

MODULE 8

DATA VISUALIZATION

Data visualization is a critical part of data science, enabling you to explore data, identify patterns, and communicate insights effectively. This module introduces two powerful Python libraries for visualization: **Matplotlib** and **Seaborn**.

By the end of this module, the reader will be able to:

Understand the Importance of Data Visualization

- Explain the role of data visualization in data science.
- Recognize how visual representations enhance data interpretation.
- Differentiate between **Matplotlib** and **Seaborn** and their use cases.

Work with Matplotlib for Basic Plotting

- Create **simple line plots** using Matplotlib.
- Customize plots by adding **titles, axis labels, legends, and grids**.
- Create several plots that are displayed together in a single figure.

Create Various Plot Types Using Matplotlib

- Make **line plots** to identify how data changes over time.
- Create **scatter plots** to explore correlation between different variables.
- Create **bar charts and histograms** to represent categorical and distribution data.
- Create **pie charts** when you have proportional or percentage data.
- Use **box plots** to evaluate a dataset's distribution as well as detect anomalous values.

Customize Matplotlib Visualizations

- Adjust **figure size, resolution, colors, and line styles**.
- Enhance readability using **annotations**.
- Create complex visualizations with **multiple axes and subplots**.

Utilize Seaborn for Enhanced Statistical Visualization

- Understand **Seaborn's built-in themes and styles** for aesthetic visualizations.
- Generate **basic Seaborn plots** such as line plots and scatter plots.

Visualize Data Using the Statistical Library Seaborn.

- Explore **distribution plots** (histograms, KDE plots, rug plots).
- Visualize **categorical data** using bar plots, box plots, and violin plots.
- Analyze **relationships** between variables using scatter plots, pair plots, and regression plots.
- Represent correlation matrices by drawing **heatmaps**.

Customize Seaborn Visualizations for Better Insights

- Modify **color palettes and themes** to improve visualization clarity.
- Combine **Seaborn with Matplotlib** for more control over figures.
- Adjust **figure size, axis labels, and plot styles** for better presentation.

Apply Data Visualization Techniques to Real-World Scenarios

- Apply these techniques to data from everyday situations (such as **stock market performance, temperature variations and the actions of consumers**).
- Put into **practice** effective strategies for sharing insights from the data..

By mastering these concepts, the reader will have the knowledge and skills necessary to **efficiently analyze and display data using visualizations**.

INTRODUCTION

Data visualization plays a vital role in the field of data science by helping analysts and researchers to investigate and communicate their findings in a clear and meaningful way. This section mainly deals with using two Python libraries for creating visualizations: **Matplotlib** and **Seaborn**.

Why Data Visualization?

1. **Helps in identifying patterns and trends**
   It's like walking into a dense forest without being able to see anything around you. There is no way to find meaningful patterns within raw data without a data visualization tool. Data visualization gives you the ability to see patterns that you would easily overlook otherwise. Viewing a line graph or scatter plot right away reveals that sales significantly increase in December and customer complaints go up after releasing a new product. Suddenly it becomes clear exactly what is happening beneath all the numbers.
2. **Simplifies complex data for better interpretation.**
   Have you ever opened a spreadsheet packed with thousands of rows and columns It is like trying to read a foreign language backwards Data visualization comes to the rescue by transforming all that chaos into clean simple visuals. A pie chart can show you in seconds how market share is divided among competitors. A heatmap can highlight problem areas at a glance. It is like turning a

tangled ball of yarn into a neat colorful braid. Suddenly complex data makes perfect sense even to someone seeing it for the first time

3. **Aids in effective decision-making.**
   When you can see your data clearly making the right decision becomes so much easier Think about it Would you rather make a decision based on a 100 page report or a clear dashboard showing exactly where profits are growing and where costs are ballooning Visualization cuts through the clutter and puts key insights right in front of you. It is like having a flashlight in a dark cave guiding you toward the right path. Whether you are launching a new product planning a marketing campaign or solving a customer issue visuals help you move faster and smarter
4. **Facilitates storytelling with data.**
   Data without a story is like a cake without frosting kind of dry and unappealing Visualization adds flavor emotion and meaning to your data turning it into a story people want to hear and remember. A rising line can tell the story of a companys growth. A colorful map can show how a movement spread across different regions. Instead of just listing facts you paint a vivid picture that captures attention and drives action. Storytelling with data is about connecting hearts and minds making numbers come alive in a way that sticks with your audience

Matplotlib vs. Seaborn

- **Matplotlib:** A powerful tool for building static, animated and interactive graphics.
- **Seaborn:** It offers a friendly and elegant interface for creating both static and statistical graphs using the popular Matplotlib library.

DATA VISUALIZATION WITH MATPLOTLIB

With Matplotlib, you can effortlessly craft static, animated and interactive plots in the Python programming language. It offers a way to generate graphs using familiar MATLAB commands.

Installing and Importing Matplotlib

Many important libraries are already installed by default in Jupyter Notebook (such as Matplotlib). If Matplotlib is not installed on your machine, use this line of code to install it:

```
pip install matplotlib
```

Importing the library:

```
import matplotlib.pyplot as plt
```

Creating Basic Plots

Line Plots

Line plots are useful for showing how things develop over the passage of time or how they vary with different values. They link each data point together through a series of lines allowing you to identify trends such as rises falls or repeating patterns. Line plots are ideal tools for monitoring trends as time goes on or in a series where the order of data has significance. They make it easy to spot differences and changes quickly.

A line plot is like a roller coaster for your data revealing every up and down along the way. Your business curves and turns with unexpected highs and lows. In an instant see triumphs setbacks and all the ups and downs along the way. Line plots turn boring numbers into an exciting journey making it fun and easy to track progress over time

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
```

```
y = [10, 20, 25, 30, 40]

# Create a line plot
plt.plot(x, y, marker='o', linestyle='-', color='b', label='Trend')
plt.title("Line Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()
plt.show()
```

Example

```
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 10, 25]
plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Basic Line Plot")
plt.grid()
plt.show()
```

Bar Charts

Bar charts allow users to compare different categories of information. They display information using rectangular bars where the length or height of each bar is proportional to the value it represents. Categories are usually placed along one axis while their values are shown on the other making it easy to see differences at a glance. Bar charts are perfect for comparing things like sales across different products survey responses or the popularity of different movie genres.

Imagine a bar chart as a friendly competition where each bar is a contestant trying to reach the top The taller the bar the bigger the bragging rights Whether it is showing which ice cream flavor rules the summer or which superhero movie crushed the box office bar charts make it super easy and super fun to spot the winners and the underdogs. With just a quick look you can cheer for the champions and spot the ones needing a little extra boost.

```
# Data
categories = ['A', 'B', 'C', 'D']
values = [15, 20, 10, 25]

# Create a bar chart
plt.bar(categories, values, color='skyblue')
plt.title("Bar Chart Example")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.show()
```

```
categories = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 8]
plt.bar(categories, values, color='purple')
plt.title("Bar Chart Example")
plt.show()
```

Scatter Plots

Scatter plots are used to show the relationship between two different variables. Each point on the plot represents an observation with its position determined by the values of those two variables. They help you spot patterns clusters outliers or any kind of trend like whether two things are moving together or not.

Now imagine you are at a lively Nigerian market like Balogun Market in Lagos or Oil Mill Market in Port Harcourt. Each trader is selling something different and their prices and number of customers vary all day long. If you plotted the price of tomatoes against the number of buyers on a scatter plot you would see little dots all over the place showing the hustle and bustle of the market. Some traders would have high prices but few customers some would have cheap prices and huge crowds and some would be right in the middle. A scatter plot captures that busy colorful energy of real life helping you quickly spot who is balling who

needs to adjust their hustle and where the sweet spot for success lies

```
# Data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a scatter plot
plt.scatter(x, y, color='red', label='Data Points')
plt.title("Scatter Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

# Simulated data: Price of tomatoes (in naira) vs Number of buyers
np.random.seed(42)  # For reproducibility

# Let's assume we have 30 traders
prices = np.random.uniform(100, 500, 30)  # Prices range from ₦100 to ₦500
buyers = n import matplotlib.pyplot as plt
import numpy as np

# Simulated data: Price of tomatoes (in naira) vs Number of buyers
np.random.seed(42)  # For reproducibility

# Let's assume we have 30 traders
prices = np.random.uniform(100, 500, 30)  # Prices range from ₦100 to ₦500
buyers = np.random.randint(5, 100, 30)    # Number of buyers ranges from 5 to 100

# Create the scatter plot
plt.figure(figsize=(9, 6))
plt.scatter(prices, buyers, color='tomato', edgecolors='black', s=100, alpha=0.7)
```

```
# Add titles and labels
plt.title('Tomato Market Hustle at Balogun/Oil Mill Market', fontsize=16)
plt.xlabel('Price of Tomatoes (₦)', fontsize=12)
plt.ylabel('Number of Buyers', fontsize=12)

# Add a grid for easier reading
plt.grid(True, linestyle='--', alpha=0.5)

# Optional: Add a little annotation for fun
for i in range(5):
    idx = np.random.randint(0, 30)
    plt.annotate('Trader {}'.format(idx+1),
                 (prices[idx], buyers[idx]),
                 textcoords="offset points",
                 xytext=(5,5),
                 ha='left',
                 fontsize=8,
                 color='green')

# Show the plot
plt.show()
```

This code generates random data for the prices of tomatoes and the number of buyers in a market. The prices are picked randomly between ₦100 and ₦500, while the number of buyers is chosen between 5 and 100. Each trader is represented as a dot on the scatter plot, with the position determined by the price of tomatoes and the number of buyers. To make the plot feel more like you're actually walking through the market, the code randomly labels a few traders, giving it a personal touch by showing who is selling what. The dots are colored **tomato** to match the vibrant, lively Nigerian tomato market vibe, adding a splash of local flavor to the visualization.

Histogram

Histograms are used to show the distribution of a set of continuous data. They look like bar charts but are a little different. Instead of comparing categories each bar groups numbers into ranges called bins and shows how many data points fall into each range. Histograms help you see things like where most values are clustered whether the data is spread out or if there are any unusual gaps or spikes.

Imagine you attend a big Nigerian wedding , you know the kind where hundreds of guests show up with plenty of jollof rice and loud music. Suppose you wanted to know the age distribution of all the guests. If you group them into age ranges like 0 to 10 years 11 to 20 years 21 to 30 years and so on a histogram would show you which age group had the most people. Maybe you will find that the 21 to 30 crew came out in full force while only a few elders sprinkled in from the 61 and above range. Just like at that owambe party the histogram quickly shows you where the crowd is gathering and where things are a little quiet

```
import numpy as np
data = np.random.randn(1000)
plt.hist(data, bins=30, color='blue', edgecolor='black')
plt.title("Histogram Example")
plt.show()
```

```
import matplotlib.pyplot as plt

# Sample data: ages of wedding guests
guest_ages = [5, 8, 12, 15, 18, 22, 25, 28, 30, 32, 35, 38, 40, 45, 48, 52, 55, 60, 65, 70, 75]

# Creating the histogram
plt.figure(figsize=(8, 5))
plt.hist(guest_ages, bins=[0, 10, 20, 30, 40, 50, 60, 70, 80], color='skyblue', edgecolor='black')
```

```
# Adding titles and labels
plt.title('Age Distribution of Guests at Nigerian Wedding')
plt.xlabel('Age Range')
plt.ylabel('Number of Guests')

# Showing the plot
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Box Plot

A box plot also called a box and whisker plot is used to show the spread and distribution of a dataset. It displays the median the upper and lower quartiles and the minimum and maximum values. In simple terms a box plot tells you where most of your data points fall where the middle value is and if there are any unusual values called outliers.

Now imagine you are comparing the prices of okra in different markets across Lagos. Some markets sell okra super cheap others super expensive and most are somewhere in between. A box plot would show you all of that at a glance. The fat box in the middle shows where most of the prices fall the line inside shows the average Lagosian price and the little whiskers stretch out to show the full range. If one market is selling okra for double the normal price it would stand out as an outlier , maybe they think they are selling gold instead of vegetables

```
plt.boxplot(data)
plt.title("Box Plot Example")
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

# Simulated data: Okra prices (in naira) across different Lagos markets
```

```
np.random.seed(42)  # For consistent results

# Let's assume 5 markets with different price spreads
market1 = np.random.normal(150, 10, 50)  # Market 1
market2 = np.random.normal(170, 15, 50)  # Market 2
market3 = np.random.normal(160, 20, 50)  # Market 3
market4 = np.random.normal(180, 5, 50)   # Market 4
market5 = np.random.normal(155, 12, 50)  # Market 5

# Combine the data
data = [market1, market2, market3, market4, market5]

# Create the box plot
plt.figure(figsize=(10, 6))
plt.boxplot(data, patch_artist=True,
            boxprops=dict(facecolor='lightgreen', color='green'),
            whiskerprops=dict(color='green'),
            capprops=dict(color='green'),
            medianprops=dict(color='red'))

# Add titles and labels
plt.title('Distribution of Okra Prices Across Lagos Markets', fontsize=16)
plt.xlabel('Markets', fontsize=12)
plt.ylabel('Okra Price (₦)', fontsize=12)
plt.xticks([1, 2, 3, 4, 5], ['Market 1', 'Market 2', 'Market 3', 'Market 4', 'Market
5'])

# Show the plot
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

This code simulates the prices of okra from five different markets, each with its own slightly different average price and price variation (spread). It then creates a colorful box plot to visualize the price distribution across these markets. The box plot clearly shows the range of prices for each market, with a red line marking the median price for each. The whiskers of the plot extend to the minimum and maximum prices, while any outliers,prices that fall

outside the whiskers,are highlighted as separate points, making it easy to spot unusually high or low prices in the data.

CUSTOMIZING PLOTS

Matplotlib gives you full control over your plots, allowing you to customize almost every aspect, from colors and labels to the overall layout. You can change the colors of lines, bars, or markers to make your plot visually appealing or match your brand or theme. You can include titles and labels on both the axes and the plot itself to help your readers easily interpret the information shown in your graph.

You can also use Matplotlib to combine several graphs in a single figure so you can easily compare each plot to one another. **Annotations** can be used to call attention to important details, so your plot is easier to understand for your viewers.

With Matplotlib, you can customize labels, layout and gridlines to produce professional and attractive visualizations of your data.

Adding Titles and Labels

Making your plot more understandable involves including titles and labels. Titles provide meaning to the entire visualization and labels clarify what each axis represents. It is simple to include titles and labels in your Matplotlib plots.

- **Title**: A title at the top of the plot helps viewers understand what the visualization shows. The title usually appears at the top of the plot.
- **Axis Labels**: Axis labels clearly indicate the data being plotted left and right. It becomes even more crucial if your plot displays data in terms of variables or units (for example, plotting "Time" on the x-axis and "Revenue (in ₦)" on the y-axis).

```
import matplotlib.pyplot as plt

# Sample data: Prices of tomatoes and number of buyers
prices = [100, 150, 200, 250, 300]
buyers = [50, 80, 60, 100, 120]

# Creating a simple scatter plot
plt.scatter(prices, buyers, color='tomato', edgecolors='black')

# Adding title and labels
plt.title('Tomato Market Prices vs. Number of Buyers', fontsize=16)
plt.xlabel('Price of Tomatoes (₦)', fontsize=12)
plt.ylabel('Number of Buyers', fontsize=12)

# Show the plot
plt.show()
```

```
plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.title("Customized Line Plot")
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.grid(True)  # Add grid lines
plt.show()
```

Multiple Plots in One Figure

Matplotlib makes it possible to arrange multiple plots in a single fi gure and offers a simpler way to compare or analyze different data sets or parts of the same data. Subplots organize your plots in a gri d layout so you can display several figures on the same graph. You have the flexibility to choose any arrangement that suits your requ irements such as a 2x2 grid or a single row of plots.

Creating Multiple Subplots:

You can generate multiple subplots by calling the plt.subplot() function or using the plt.subplots() method which arranged and labels the axes.

EXAMPLE

```
import matplotlib.pyplot as plt
import numpy as np

# Sample data: Random values for different plots
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)

# Create a 2x2 grid of subplots
fig, axs = plt.subplots(2, 2, figsize=(10, 8))  # 2 rows, 2 columns

# First subplot - Sine wave
axs[0, 0].plot(x, y1, color='blue')
axs[0, 0].set_title('Sine Wave')
axs[0, 0].set_xlabel('X')
axs[0, 0].set_ylabel('sin(x)')

# Second subplot - Cosine wave
axs[0, 1].plot(x, y2, color='green')
axs[0, 1].set_title('Cosine Wave')
axs[0, 1].set_xlabel('X')
axs[0, 1].set_ylabel('cos(x)')

# Third subplot - Tangent wave
axs[1, 0].plot(x, y3, color='red')
axs[1, 0].set_title('Tangent Wave')
axs[1, 0].set_xlabel('X')
axs[1, 0].set_ylabel('tan(x)')
axs[1, 0].set_ylim(-10, 10)  # Limit y-axis for better visualization

# Fourth subplot - Empty plot or another example
axs[1, 1].text(0.5, 0.5, 'No Plot', fontsize=15, ha='center', va='center')
axs[1, 1].set_title('Empty Plot')

# Adjust layout to prevent overlap
plt.tight_layout()
```

```
# Show the plot
plt.show()
```

EXAMPLE

```
# Create subplots
fig, ax = plt.subplots(1, 2, figsize=(10, 5))

# Plot 1
ax[0].plot(x, y, color='blue')
ax[0].set_title("Line Plot")

# Plot 2
ax[1].scatter(x, y, color='red')
ax[1].set_title("Scatter Plot")

plt.show()
```

Annotations

Annotations in Matplotlib let you place text, arrows or other shapes directly on your plot at the exact locations you want. This feature allows you to call attention to crucial information, explain details or make summaries right on your graph. You have the power to highlight a variety of elements, like exceptional values or significant occurrences which significantly improves the clarity and impact of what is being shown by your plots.

Common Uses for Annotations:

- **Highlighting data points**: Adding a label to an outlier or a significant point in the graph.
- **Explaining trends**: Marking where a particular change happens in a curve or a key event.
- **Pointing out values**: Labeling specific data points with their values for better clarity.

- **Adding arrows**: You can sketch arrows that directly highlight the exact points of interest in your plot.

Example

```
import matplotlib.pyplot as plt
import numpy as np

# Sample data: Sine wave
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create the plot
plt.plot(x, y, label='Sine Wave', color='blue')

# Annotating a specific point on the sine wave
max_point = (np.pi/2, np.sin(np.pi/2))  # Point where sine reaches maximum (π/2, 1)
plt.annotate('Max Point (π/2, 1)',
        xy=max_point,
        xytext=(max_point[0] + 1, max_point[1] - 0.5),  # Text position
        arrowprops=dict(facecolor='red', arrowstyle='->'),  # Arrow properties
        fontsize=12, color='black')

# Annotating another point (0, 0) for example
plt.annotate('Origin (0, 0)',
        xy=(0, 0),
        xytext=(1, -0.5),
        fontsize=12, color='green')

# Adding titles and labels
plt.title('Sine Wave with Annotations', fontsize=16)
plt.xlabel('X Axis', fontsize=12)
plt.ylabel('Y Axis', fontsize=12)

# Show the plot
plt.legend()
```

```
plt.show()
plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.annotate('Peak', xy=(4, 30), xytext=(3, 35),
        arrowprops=dict(facecolor='black', shrink=0.05))
plt.title("Annotated Plot")
plt.show()
```

Example

```
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 10, 25]
plt.scatter(x, y, color='r')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Scatter Plot Example")
plt.show()


fig, axes = plt.subplots(1, 2, figsize=(10, 4))
axes[0].plot(x, y, color='g')
axes[0].set_title("First Plot")
axes[1].scatter(x, y, color='r')
axes[1].set_title("Second Plot")
plt.show()
```

DATA VISUALIZATION WITH SEAVORN

Seaborn provides high-quality data visualization capabilities through an interface that is built on top of Matplotlib. It is focused on producing **statistical graphics** and allows users to generate visually pleasing and informative plots easily. Seaborn enables users to generate sophisticated visualizations quickly and easily by focusing on **pandas DataFrame** data.

Key Features of Seaborn:

1. **Built-in Themes and Styles**:
   Seaborn features a variety of theme and color options to make professional-looking plots quickly.

2. **Statistical Plots**:
   It can produce advanced graphical methods such as violin plots, box plots and regression plots, making it well-suited for performing data analysis tasks.
3. **Integration with Pandas**:
   Seaborn integrates easily with pandas DataFrames, enabling users to quickly visualize information from within their DataFrames.
4. **Simplified Syntax**:
   Seaborn allows you to produce sophisticated plots with fewer lines of code than Matplotlib, making plotting easier.
5. **Support for Multi-plot Grids**:
   Seaborn features tools such as FacetGrid and PairGrid , allowing for the generation of multivariate plots that can be used to understand patterns in the data.

Seaborn is often employed for:

- *Visualizing distributions (e.g., histograms, KDE plots).*
- *Exploring relationships between variables (e.g., scatter plots, pair plots).*
- *Comparing groups or categories (e.g., bar plots, box plots).*
- *Visualizing correlations (e.g., heatmaps)*

Given the Tip Dataset in table 8-1, Let's create a scatter plot using sns

Table 8-1: Tip Dataset table

s/n	total_bill	Tip	sex	smoker	day	time	Size
1	16.99	1.01	Female	No	Sun	Dinner	2
2	10.34	1.66	Male	No	Sun	Dinner	3
3	21.01	3.5	Male	No	Sun	Dinner	3

4	23.68	3.31	Male	No	Sun	Dinner	2
5	24.59	3.61	Female	No	Sun	Dinner	4
6	25.29	4.71	Male	No	Sun	Dinner	4
7	8.77	2.0	Male	No	Sun	Dinner	2
8	26.88	3.12	Male	No	Sun	Dinner	4
9	15.04	1.96	Male	No	Sun	Dinner	2
10	14.78	3.23	Male	No	Sun	Dinner	2

```
import seaborn as sns
import matplotlib.pyplot as plt
# Load example dataset
tips = sns.load_dataset("tips")
# Create a scatter plot
sns.scatterplot(x="total_bill", y="tip", data=tips)
plt.title("Total Bill vs Tip")
plt.show()
```

Installing and Importing Seaborn

Jupyter Notebook is already equipped with a wide range of popular libraries including seaborn. To install seaborn if it is not pre-installed on your computer, input the following command:

pip install seaborn

Importing the library:

import seaborn as sns

Setting Seaborn Themes

sns.set_style("darkgrid")

STATISTICAL DATA VISUALIZATION WITH SEABORN

Heatmaps for correlation analysis

Imagine you're at a party, and you're trying to figure out which people are vibing with each other the most; like, who's having the time of their life on the dance floor. Now, instead of guessing, you've got a cool tool to give you the lowdown: a **heatmap!**

When you use a heatmap for **correlation analysis**, it's like a *party tracker* for numbers. You're not just looking at random data points; you're getting a bird's eye view of how everything is connected.

In simple terms, heatmaps visually represent the relationship between variables, using colors. The darker the color, the stronger the connection, while lighter shades mean the relationship is weak or non-existent. For example, in **predictive maintenance** (a hot topic!), you could use a heatmap to figure out which sensors in your machines are most strongly linked,like when you notice your oil pressure and temperature sensors always dance in sync. So, the next time you see one of those colorful grids with reds, greens, or blues popping up, just remember: it's like the data's way of showing you who's cool with who, so you can make smarter decisions without any guesswork!

```
import seaborn as sns
import numpy as np

# Create a correlation matrix
data = np.random.rand(5, 5)

# Create a heatmap
sns.heatmap(data, annot=True, cmap='coolwarm')
plt.title("Heatmap Example")
plt.show()
```

```
# Import Seaborn
import seaborn as sns
import pandas as pd

# Sample dataset - You can replace this with your own data
data = {
   'sensor_1': [1, 2, 3, 4, 5],
   'sensor_2': [5, 4, 3, 2, 1],
   'sensor_3': [2, 3, 4, 5, 6],
```

```
    'sensor_4': [5, 6, 7, 8, 9]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Compute the correlation matrix
corr_matrix = df.corr()

# Create and display the heatmap using only Seaborn
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f',
linewidths=0.5, square=True)

# Show the plot
sns.plt.show()
```

Pairplots

Pairplots are an excellent way to visualize relationships between multiple numerical variables. In Seaborn, the sns.pairplot() function creates a grid of scatterplots for each pair of features, along with histograms or density plots along the diagonal to show distributions.

Think of a pairplot like a party where different groups of people (your data variables) are mingling. You've got several sensor readings.Let's say sensor_1, sensor_2, etc.,and you're curious about how they interact with each other. Instead of guessing, a pairplot comes to the rescue by plotting a grid of scatter plots, showing you exactly how each pair of sensors relates. If two sensors are in sync, their scatter plot will form a clear line or pattern. If they're not, you'll see more scattered dots. The diagonal of this grid is where each sensor hangs out on its own, showing you how it behaves over time with histograms or density plots.

This is super useful, especially when you're dealing with complex data from sensors in industries like oil and gas. For example, a pairplot can quickly reveal if temperature and pressure sensors are working hand in hand (likely highly correlated) or if there's no real relationship between vibration and humidity sensors. What's great is that pairplots also allow you to see these relationships visually in one glance, saving you from having to analyze each pair of sensors individually.

It's like having a savvy friend at the party who knows exactly who's mixing well with who, without needing to ask everyone. So, instead of stressing over how your sensors interact, the pairplot gives you an easy-to-understand, colorful grid that shows all the connections in your data.

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample dataset - You can replace this with your own data
data = {
    'sensor_1': [1, 2, 3, 4, 5],
    'sensor_2': [5, 4, 3, 2, 1],
    'sensor_3': [2, 3, 4, 5, 6],
    'sensor_4': [5, 6, 7, 8, 9]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create a pairplot of the dataset using Seaborn
sns.pairplot(df)

# Add title
plt.suptitle("Pairplot of Sensor Data", y=1.02)
```

```
# Show the plot
plt.show()
```

```
# Load a sample dataset
iris = sns.load_dataset('iris')

# Create a pairplot
sns.pairplot(iris, hue='species')
plt.title("Pairplot Example")
plt.show()
```

Density Plot

A density plot is a smooth, continuous representation of the distribution of data, much like a histogram, but without the bins. It's used to show the probability density function (PDF) of a continuous random variable, so you get a smooth curve instead of the blocky look you get with histograms. Think of it as an elegant way of seeing how your data is spread out, with high peaks showing where most of your data points are concentrated and valleys indicating less frequent data.

Imagine a density plot like a smooth Afrobeat groove,it shows the smooth flow of your data from left to right, highlighting where the data "moves" or "clusters" the most. If your data was a party, the density plot is like showing you the hotspots where everyone's gathered and the quieter corners where not much is happening.

In Python, you can easily create a density plot using Seaborn's `sns.kdeplot()`. This is especially helpful when you want to visualize the distribution of individual variables or compare distributions across multiple variables.

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
```

```
# Sample dataset - You can replace this with your own data
data = {
   'sensor_1': [1, 2, 3, 4, 5],
   'sensor_2': [5, 4, 3, 2, 1],
   'sensor_3': [2, 3, 4, 5, 6],
   'sensor_4': [5, 6, 7, 8, 9]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create a density plot for a single variable using Seaborn
sns.kdeplot(df['sensor_1'], shade=True, color='blue', label='Sensor 1')

# Optionally, add more density plots for other sensors
sns.kdeplot(df['sensor_2'], shade=True, color='green', label='Sensor 2')

# Add title and labels
plt.title("Density Plot of Sensor Data")
plt.xlabel("Sensor Readings")
plt.ylabel("Density")

# Show the legend and plot
plt.legend()
plt.show()
```

Violin Plots

Imagine you're at a party and you see a huge group of people dancing. Some are doing the *Shaku Shaku* in one corner, while others are chilling with some Afrobeat moves in another. Now, instead of guessing who's dancing better or how many people are jamming in each group, you get this funky violin plot to give you the full vibe!

In simpler terms, a violin plot is like a combination of a box plot and a density plot. It shows you the distribution of your data, like

how your sensor readings are spread out. The wider the "violin," the more data points (or people) are hanging out in that range. The "stem" in the middle is the line that shows you the range of your data, and the "bulge" or wider part is where most of your data points are concentrated, just like where everyone is crowded at the party!

What makes it even more fun is that a violin plot lets you compare multiple variables at once. For example, you could check how temperature varies compared to pressure,are they dancing the same rhythm, or are they in different parts of the party? It gives you a quick, clear visual of the spread of your data,whether they're partying together or off doing their own thing.

So, if you want to know how sensor data is behaving without digging deep into boring numbers, a violin plot gives you the whole vibe, letting you see the highs, lows, and everything in between, all in one smooth, colorful plot. It's like seeing the dancefloor from above,no more guessing!

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample dataset - You can replace this with your own data
data = {
   'sensor_1': [1, 2, 3, 4, 5],
   'sensor_2': [5, 4, 3, 2, 1],
   'sensor_3': [2, 3, 4, 5, 6],
   'sensor_4': [5, 6, 7, 8, 9]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create a violin plot using Seaborn
sns.violinplot(data=df)
```

```
# Add title
plt.title("Violin Plot of Sensor Data")

# Show the plot
plt.show()
```

Histogram and KDE Plot

A **Histogram** is like a fun way of grouping your data into different "bins," similar to how you'd divide a group of people into age ranges at a party. Imagine you're counting how many people fall into age groups, like 20-25 years, 25-30 years, and so on. Each **bar** in a histogram represents how many data points (or people, in this example) fall into each specific range. So, if the bar is taller, it means more people are in that age range. It's perfect for showing the frequency or count of your data within specific intervals, making it easy to see which range is most common or "popular."

On the other hand, a **KDE Plot** takes things a step further by smoothing out those bars into a continuous **curve**. Instead of jagged, discrete bars, the KDE plot gives you a smooth line that illustrates the overall flow or distribution of your data. It's like saying, "Yes, we know this age group is common, but let's show the general trend of the data as a smooth, flowing curve." The smooth curve lets you see where the data is concentrated and how it tapers off, making it easier to understand the underlying trend without all the sharp edges.

Combining histograms and KDE plots lets you see both the strength and subtlety of your data. A **histogram** lets you see exactly how many data points fall into various categories, whereas the **KDE plot** shows you the **smooth** pattern of the data. Combining them is similar to viewing both the crowd at a party

(as shown by the histogram) and the seamless dancing that results from the coordination of everyone's movements (shown by the KDE plot). They provide a deeper understanding of how your data fluctuates.

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample dataset - You can replace this with your own data
data = {
   'sensor_1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 5, 6]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create the Histogram and KDE Plot for sensor_1
sns.histplot(df['sensor_1'], kde=True, color='blue', bins=5)

# Add title and labels
plt.title("Histogram and KDE Plot of Sensor 1")
plt.xlabel("Sensor Readings")
plt.ylabel("Frequency / Density")

# Show the plot
plt.show()
```

Scatter Plot with Regression Line

You can see the connection between two variables by using a scatter plot with a regression line. The scatter plot shows individual data points on a two-dimensional graph, where the x-axis and y-axis represent different variables. The regression line (or line of best fit) is drawn through the points to show the overall

trend or relationship between these variables, helping you see if there's a positive or negative correlation.

For example, imagine you're analyzing the relationship between temperature and pressure in a machine. The scatter plot will show how each pair of temperature and pressure values relate, and the regression line will help you see if higher temperatures are associated with higher pressure (or vice versa). The regression line is the best linear approximation of the relationship between the variables.

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample dataset - You can replace this with your own data
data = {
   'temperature': [22, 24, 25, 26, 28, 30, 31, 33, 34, 35],
   'pressure': [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create a scatter plot with a regression line
sns.regplot(x='temperature', y='pressure', data=df, color='green')

# Add title and labels
plt.title("Scatter Plot with Regression Line: Temperature vs Pressure")
plt.xlabel("Temperature (°C)")
plt.ylabel("Pressure (hPa)")

# Show the plot
plt.show()
```

REAL-WORLD APPLICATIONS OF DATA VISUALIZATION

Case Study 1: Stock Market Data Visualization

```
stock_data = pd.read_csv("stocks.csv")
sns.lineplot(x='Date', y='Close', data=stock_data)
plt.xticks(rotation=45)
plt.title("Stock Market Trends")
plt.show()
```

Case Study 2: Customer Sales Analysis

```
sales_data = pd.read_csv("sales.csv")
sns.barplot(x='Product', y='Sales', data=sales_data)
plt.xticks(rotation=45)
plt.title("Product Sales Analysis")
plt.show()
```

QUESTIONS

1. What is the difference between Matplotlib and Seaborn?
2. Explain the purpose of a heatmap.
3. What is the difference between a histogram and a box plot?
4. What is the difference between a box plot and a violin plot?
5. Create a line plot to visualize the growth of a company's revenue over 5 years.\
6. Use a bar chart to compare the population of 5 cities.
7. Create a scatter plot to visualize the relationship between hours studied and exam scores.
8. Customize a plot by adding a title, labels, and grid lines.
9. Create a figure with two subplots: one line plot and one scatter plot.
10. Create a heatmap to visualize the correlation between features in the Iris dataset.
11. Use a pairplot to explore relationships between numerical variables in the Titanic dataset.
12. Create a violin plot to compare the distribution of sepal lengths across different species in the Iris dataset

MODULE 9

LINEAR ALGEBRA FOR DATA SCIENCE

In this module, we covered key linear algebra concepts such as vectors, matrices, determinants, eigenvalues, and singular value decomposition, all with Python implementations. Mastering these concepts is crucial for understanding and implementing machine learning algorithms.

By the end of this module, the reader will be able to:

- Understand the fundamental concepts of linear algebra and its importance in data science.
- Define and perform operations on vectors using NumPy.
- Implement vector addition and scalar multiplication in Python.
- Define and manipulate matrices, including matrix addition and multiplication.
- Compute the determinant and inverse of a matrix using NumPy.
- Understand the significance of eigenvalues and eigenvectors in data science.
- Compute eigenvalues and eigenvectors in Python.
- Perform Singular Value Decomposition (SVD) for dimensionality reduction and recommendation systems.
- Apply linear algebra techniques in machine learning and data science tasks such as:
    - **Dimensionality Reduction** (PCA using eigenvalues and eigenvectors).
    - **Regression Models** (Matrix operations in linear regression).

- **Neural Networks** (Weights and activations as matrices).
- **Recommendation Systems** (Using SVD for collaborative filtering).

INTRODUCTION TO LINEAR ALGEBRA

Linear algebra is a foundational mathematical discipline in data science It provides essential tools for handling datasets performing transformations and optimizing machine learning algorithms In this module we will cover fundamental linear algebra concepts and their applications in data science using Python

Think of linear algebra as the **language of data** Whenever we are working with huge amounts of information whether it is numbers from sensors images videos or even texts we usually arrange them neatly into **vectors** and **matrices** Linear algebra gives us the grammar and rules to manipulate these structures making it possible to perform calculations efficiently and uncover hidden patterns inside the data

From basic operations like adding two vectors to more advanced techniques like matrix multiplication eigenvalues and singular value decomposition linear algebra is everywhere Without it algorithms such as Principal Component Analysis PCA for dimensionality reduction or deep learning models like neural networks would simply not work

In this module we will not only study the theory but also **practically code** the key operations using Python libraries like **NumPy** By the end you will see linear algebra not as some boring abstract maths but as a powerful toolbox that makes real world data problems much easier to solve

VECTORS AND VECTOR OPERATIONS

A vector is simply a list of numbers arranged in a specific order It is one of the most important concepts in linear algebra In the real world you can think of a vector as a way to represent things like a direction and a distance For example if you are giving someone instructions to find your house you might say Walk three steps forward and then two steps to the right This instruction can be written neatly as a vector 3 2

In data science vectors are used to represent all kinds of data A row in a spreadsheet where you record things like age salary and years of experience for a person can be thought of as a vector Each number in the vector is called an element or a component

A vector is considered a one-dimensional array that represents magnitude and direction. \

Defining Vectors in Python

In Python we often use lists to represent simple vectors but for real data science work it is better to use NumPy arrays NumPy is a powerful library that makes mathematical operations easier and faster

To define a vector using NumPy you first import the library by writing import numpy as np Then you can create a vector by using the np dot array function

```
import numpy as np
v = np.array([2, 3, 5])
print(v)
```

Vector Addition and Scalar Multiplication

Vector addition is simply the process of adding two vectors together To add two vectors you add their corresponding elements.

Scalar multiplication means multiplying a vector by a single number called a scalar

Example

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
s = 2

# Vector Addition
v_sum = v1 + v2
print("Vector Sum:", v_sum)

# Scalar Multiplication
v_scaled = s * v1
print("Scaled Vector:", v_scaled)
```

MATRICES AND MATRIX OPERATIONS

Imagine you are at a Nigerian market and you are tracking how much you spent on tomatoes peppers and onions across three different stalls You can organize all this information neatly into a table with rows and columns That table is exactly what a matrix looks like A matrix is simply a two dimensional arrangement of numbers where each row can represent one stall and each column can represent an item you are buying

In data science matrices are everywhere We use them to represent full datasets where each row could be one customer and each column could be information about that customer like their age income and number of purchases Matrices help us keep everything organized and ready for mathematical operations

Matrix operations are like the calculations you do when you want to find out your total spending or compare prices across stalls You can add two matrices together if they are the same size by adding their matching elements one by one You can multiply a matrix by a number to scale all the values up or down just like when you are doubling your shopping list before Christmas and you know everything will cost twice as much

Another important operation is matrix multiplication which is a little more complex It is like when you are combining information from two different tables to get a final result Maybe you have one table for the price of each item and another for the quantity you bought Matrix multiplication helps you combine these tables in a smart way to find your total spending automatically

In short matrices are not just scary grids of numbers They are powerful friends that help you manage and transform data whether you are solving business problems building machine learning models or even planning your next shopping trip.

Defining Matrices in Python

```
A = np.array([[1, 2], [3, 4]])
print(A)
```

Matrix Addition and Multiplication

```
B = np.array([[5, 6], [7, 8]])
# Matrix Addition
C = A + B
print("Matrix Sum:\n", C)
# Matrix Multiplication
D = np.dot(A, B)
print("Matrix Product:\n", D)
```

Determinant and Inverse of a Matrix

Now let us talk about two very important things in the world of matrices the determinant and the inverse Think of the

determinant like the signature of a matrix It is just one number but it tells you a lot about the matrix If the determinant is zero it means the matrix is somehow not powerful enough to do some operations just like a car without fuel cannot move But if the determinant is not zero then the matrix is full of energy and ready to solve problems

The inverse of a matrix is like the opposite of the matrix Imagine you are trying to undo a move in a game If moving forward takes you to a new spot then moving backward should bring you back to where you started That backward move is like the inverse In mathematics if you multiply a matrix by its inverse you get something called the identity matrix which is like pressing the reset button

In data science and machine learning finding the inverse of a matrix is super important especially when you are solving systems of linear equations For example when you want to predict house prices based on different factors you often have to solve such systems Matrices their determinants and inverses make the whole process smooth and possible

So whenever you hear determinant or inverse just know you are dealing with the secret powers that tell us whether a matrix is ready to help solve problems or not

Computing the Determinant

To calculate the determinant of a matrix in Python we make life very easy by using NumPy All you have to do is import the det function from numpy dot linalg Once you have your matrix A ready you just call det A and it will give you the determinant straight This saves you from doing long manual calculations that can make your head spin like NEPA fan When you print the result you will see one special number that tells you the full story of the matrix whether it is strong or weak

```
from numpy.linalg import det
determinant = det(A)
print("Determinant:", determinant)
```

Computing the Inverse

Finding the inverse of a matrix is just as easy thanks to NumPy You import the inv function from numpy dot linalg and then simply call inv A on your matrix If the matrix has an inverse NumPy will calculate it sharp sharp and give you a new matrix which is the inverse Printing it will show the values nicely arranged If your matrix does not have an inverse maybe because the determinant is zero then NumPy will also complain and let you know

With just a few lines of code you can handle serious matrix problems that used to take people one full day with pen and paper

```
from numpy.linalg import inv
A_inv = inv(A)
print("Inverse Matrix:\n", A_inv)
```

EIGENVALUES AND EIGENVECTORS

Now let us talk about eigenvalues and eigenvectors which sound big but are actually very powerful ideas in data science Imagine you are standing in a busy Lagos market and everybody is moving in different directions Suddenly you notice that no matter the crowd one narrow path stays steady and people naturally move along it That steady path is like an eigenvector and the strength of movement along it is like the eigenvalue

In simple terms eigenvectors show us special directions where things do not really change direction when you apply a transformation while eigenvalues tell us how much things stretch or shrink along those directions In data science especially in Principal Component Analysis PCA eigenvalues and eigenvectors

help us find the most important patterns in our data Instead of carrying heavy bags of data with hundreds of features PCA helps us use only the most important ones and throw the rest away
So anytime you hear eigenvalues and eigenvectors just know that they are the secret shortcuts that make it easy to understand big complicated datasets without stressing yourself too much.

Computing Eigenvalues and Eigenvectors

To compute eigenvalues and eigenvectors in Python we turn to NumPy which makes it super easy First you need to import the eig function from numpy dot linalg This function does the magic for you When you apply it to a matrix A it returns two important things: the eigenvalues and the eigenvectors The eigenvalues will tell you how much the data stretches or shrinks along the special directions and the eigenvectors give you those special directions themselves
Once you call eig A it will give you the eigenvalues and eigenvectors as two separate results You can then print them to see which directions are the most important in your data and how much they stretch or shrink In real life this is like saying which roads in the market are the busiest and which one stretches the most through the crowd

```
from numpy.linalg import eig
eigenvalues, eigenvectors = eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

SINGULAR VALUE DECOMPOSITION (SVD)

Singular Value Decomposition or SVD might sound like something from a science fiction movie but trust me it is one of the most powerful tools in data science SVD is a way of breaking down a matrix into three smaller matrices that are much easier to

work with It's like taking a big complicated dish and breaking it into smaller more manageable pieces so you can enjoy the meal in the right way

SVD is used everywhere especially when you want to simplify complex data For example in recommendation systems like Netflix or YouTube SVD helps to find patterns in how users interact with content and predict what they might like next In dimensionality reduction SVD helps reduce the number of features in your data while keeping the important information intact It's like when you want to reduce the number of actors in a movie but still make sure the plot remains interesting and engaging

SVD works by decomposing your original matrix A into three smaller matrices U S and V where U and V are orthogonal matrices and S is a diagonal matrix These three matrices can then be used to perform things like data compression and noise reduction making SVD an important part of any data scientist's toolkit

Performing SVD in Python

Now that we know what SVD is all about let's see how to do it in Python with NumPy The good news is that Python and NumPy make it super simple You only need to use the svd function from numpy dot linalg and just like that you can perform Singular Value Decomposition on your matrix A

```
from numpy.linalg import svd
U, S, Vt = svd(A)
print("U Matrix:\n", U)
print("Singular Values:", S)
print("V Transpose Matrix:\n", Vt)
```

APPLICATIONS OF LINEAR ALGEBRA IN DATA SCIENCE

APPLICATIONS OF LINEAR ALGEBRA IN DATA SCIENCE

The knowledge of Linear Algebra is required in your data science journey. Below are some key applications of linear algebra in data science:

1. Dimensionality Reduction

High-dimensional datasets can be challenging to process and analyze. Linear algebra techniques help reduce the dimensionality while retaining meaningful information.

Principal Component Analysis (PCA)

PCA is a popular dimensionality reduction technique that uses **eigenvectors and eigenvalues** to transform high-dimensional data into a lower-dimensional space.

Example: Applying PCA in Python

```
from sklearn.decomposition import PCA
import numpy as np

# Sample dataset (5 samples, 3 features)
data = np.array([[2.5, 2.4, 3.5],
                 [0.5, 0.7, 1.1],
                 [2.2, 2.9, 3.1],
                 [1.9, 2.2, 2.8],
                 [3.1, 3.0, 4.0]])

# Apply PCA to reduce dimensions to 2
pca = PCA(n_components=2)
transformed_data = pca.fit_transform(data)

print("Reduced Data:\n", transformed_data)
```

2. Regression Models (Linear Regression)

$$\boldsymbol{\theta} = (\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}$$

Where:

- $\boldsymbol{\theta}$ is the vector of model parameters (coefficients) we need to compute.
- $\boldsymbol{X}$ is the feature matrix (your input data, where each row represents a data point and each column represents a feature).
- $\boldsymbol{X}^{\boldsymbol{T}}$ is the transpose of the feature matrix $\boldsymbol{X}$
- $(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}$ is the inverse of the matrix $\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$
- $\boldsymbol{y}$ is the target vector (the actual values you are trying to predict).

Linear regression is one of the oldest and most reliable tools in the world of data science It is a statistical method used to predict a target value based on input features or independent variables In simple terms if you wanted to predict the price of a house based on its size number of rooms and age you could use linear regression to draw the best-fitting line that connects the dots of data points to make the best prediction

Now the magic behind linear regression is heavily dependent on matrix operations These operations help us find the perfect line that minimizes the difference between the predicted values and the actual data points

To compute θ the formula relies on matrix multiplication and inversion These matrix operations allow us to find the values of θ that best fit the data by minimizing the errors in prediction This is what we call solving for the best-fitting line

In real-life data science applications using this formula allows us to make predictions based on the patterns in the data whether we are predicting house prices stock market trends or any other kind of continuous data

Example in Python

```
import numpy as np

# Example data: X is the feature matrix, y is the target vector
# Let's assume we have 3 data points and 2 features
X = np.array([[1, 1], [1, 2], [2, 2]])  # Add a column of 1s for the intercept term
y = np.array([1, 2, 2])

# Add a column of 1s to X to account for the intercept (bias term)
X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Compute theta using the Normal Equation: θ = (X^T X)^-1 X^T y
theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

print("Theta (Model Parameters):", theta)
```

3. Neural Networks

Neural networks, which are a key part of machine learning, rely heavily on matrices for their computations. The network itself is made up of layers of nodes (neurons), and these nodes are connected by weights that determine how data flows through the network. These weights are stored in weight matrices, which are crucial in transforming inputs into predictions. When an input is passed through the network, matrix multiplication takes place between the input data and the weight matrices to generate the outputs for each layer.

Activation functions are then applied to the results of these matrix multiplications to introduce non-linearity into the model, making it capable of learning complex patterns. These activation functions, such as sigmoid or ReLU, are element-wise operations that are applied to each value in the matrix of outputs from the previous layer.

Backpropagation, which is how neural networks learn, also involves matrices. It's the process by which the network adjusts its

weights based on the error between predicted and actual outputs. During backpropagation, the gradients of the loss function with respect to the weight matrices are calculated, and this involves matrix differentiation. These gradients tell the network how to update the weights to reduce the error, and the process is repeated for each layer, refining the model's parameters over time.

Example: Simple Matrix-Based Forward Propagation

```
# Inputs and weights (as matrices)
inputs = np.array([[0.5, 0.2]])
weights = np.array([[0.3, 0.8], [0.5, 0.1]])
bias = np.array([0.1, 0.2])

# Compute the output
output = np.dot(inputs, weights) + bias
print("Output:", output)
```

Example

This example demonstrates forward propagation (input through the network) and the basic structure for a neural network layer.

```
Import numpy as np
# Example Neural Network with 1 hidden layer

# Input data (4 samples, 3 features)
X = np.array([[0, 0, 1],
              [1, 0, 1],
              [0, 1, 1],
              [1, 1, 1]])

# Target outputs (4 samples, 1 output)
y = np.array([[0], [1], [1], [0]])

# Random weight matrices for input to hidden layer (3 input features -> 4 neurons in hidden layer)
np.random.seed(42)
```

```
weights_input_hidden = np.random.rand(X.shape[1], 4)

# Random weight matrix for hidden to output layer (4 hidden neurons -> 1 output)
weights_hidden_output = np.random.rand(4, 1)

# Activation function (Sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Sigmoid derivative (used in backpropagation)
def sigmoid_derivative(x):
    return x * (1 - x)

# Forward propagation
hidden_layer_input = np.dot(X, weights_input_hidden)  # Input layer to hidden layer multiplication
hidden_layer_output = sigmoid(hidden_layer_input)  # Apply activation function

output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)  # Hidden layer to output layer multiplication
output_layer_output = sigmoid(output_layer_input)  # Apply activation function

print("Predicted Output:")
print(output_layer_output)

# Let's assume we're training and now want to compute the error using backpropagation
error = y - output_layer_output
print("Error:")
print(error)

# Backpropagation
output_layer_delta = error * sigmoid_derivative(output_layer_output)  # Gradient of the loss with respect to output layer
```

```
hidden_layer_error = output_layer_delta.dot(weights_hidden_output.T) # Error propagated to hidden layer
hidden_layer_delta = hidden_layer_error * sigmoid_derivative(hidden_layer_output) # Gradient of the loss with respect to hidden layer

# Update weights using the deltas
learning_rate = 0.1
weights_hidden_output += hidden_layer_output.T.dot(output_layer_delta) * learning_rate
weights_input_hidden += X.T.dot(hidden_layer_delta) * learning_rate

print("\nUpdated Weights:")
print("Weights (Input -> Hidden):")
print(weights_input_hidden)
print("Weights (Hidden -> Output):")
print(weights_hidden_output)
```

4. Recommendation Systems

Recommendation systems are a powerful tool used by platforms like Netflix, Amazon, and YouTube to suggest products, movies, or content to users. They use information on user preferences to offer personalized suggestions. **Singular Value Decomposition** is a major practice used when developing recommendation systems.

In a recommendation system, the process often begins by creating a **user-item interaction matrix** which uses a row for each user and a column for each item. A ranking is given for each item depending on how much the user has viewed and used it. Nonetheless, many entries in the matrix are empty since the same user does not interact with every item. SVD is used here for a good reason. A SVD approach to the user-item matrix results in three smaller matrices called U, S and V^T which outline the main trends in what users prefer and what each item has in common.

This is how it works:

- U: Consists of features that are unique to each user (latent factors).
- S: Contains singular values which show how much influence a variable has in the results.
- V^T: Includes special features that are important only to specific items.

After extracting these factors, we can estimate the missing entries such as how a user could rate a movie.

```
import numpy as np
from numpy.linalg import svd

# Sample user-item interaction matrix (rows = users, columns = items)
# A value of 0 indicates that the user has not interacted with the item yet
R = np.array([
   [5, 0, 0, 1, 0],
   [4, 0, 0, 1, 0],
   [1, 1, 0, 0, 0],
   [1, 0, 0, 4, 0],
   [0, 1, 5, 4, 0],
])

# Step 1: Perform SVD on the matrix R
U, S, Vt = svd(R, full_matrices=False)

# Step 2: Choose the number of singular values to keep (e.g., k = 2)
k = 2  # We reduce the matrix to 2 dimensions
U_k = U[:, :k]
S_k = np.diag(S[:k])
Vt_k = Vt[:k, :]

# Step 3: Reconstruct the approximation of the original matrix R
R_approx = np.dot(np.dot(U_k, S_k), Vt_k)

print("Original User-Item Matrix (R):")
print(R)
print("\nApproximated User-Item Matrix (R_approx):")
print(R_approx)
```

```
# Step 4: Predict missing values in the user-item interaction matrix
# Let's assume the value at (0, 1) was missing (0 in original matrix)
predicted_rating = R_approx[0, 1]
print("\nPredicted Rating for User 0 and Item 1:", predicted_rating)

# Sample user-item rating matrix
ratings = np.array([[5, 4, 0, 1],
```

Example: Applying SVD for Matrix Factorization

```
from numpy.linalg import svd
                [4, 0, 4, 2],
                [3, 5, 3, 0],
                [0, 3, 4, 5]])

# Apply SVD
U, S, Vt = svd(ratings)

print("Singular Values:", S)
```

5. Anomaly Detection

Anomaly detection is the process of identifying rare items, events, or observations that deviate significantly from the majority of the data. These outliers can provide valuable insights, such as detecting fraud in financial transactions, identifying faulty equipment in predictive maintenance, or spotting unusual behavior in network trafficIn the process, linear algebra is particularly important for technology like Mahalanobis Distance.

The Mahalanobis Distance is used to find the distance from a single point to an entire distribution. Mahalanobis Distance is different from Euclidean Distance because it includes the effect of data correlations and scales the distance based on the data values' variability. As a result, it becomes very handy for finding outliers in sets of data where features share a connection.

The Mahalanobis Distance is defined as: $D_M(x) = \sqrt{(x-\mu)^T \sum^{-1}(x-\mu)}$

Where:

- x : is the data point we are testing to see how it affects the outcome..
- μ is the average number found in the dataset.
- $\sum^{-1} is$ the inverse of the covariance matrix for the data is called the precision matrix.
- $D_M(x)$ gives the Mahalanobis distance and expresses to what extent the data point xxx is far from the mean μ, using the standard deviation of the observations.

The main strength of the Mahalanobis Distance lies in its ability to handle data that has features that are related to each other. A point with a greater Mahalanobis distance tends to be far from the mean and could mark it as an outlier.

How to Use Mahalanobis Distance for Anomaly Detection:

1. **Determine the mean vector μ of the data set.**
2. **Find the covariance matrix Σ that describes the dataset..**
3. **Calculate the Mahalanobis Distance for all data points in the data set.**
4. **Thresholding**: Any points with a higher Mahalanobis distance than a given threshold are recognized as outliers.

Use Case:

In a **credit card fraud detection system**, for instance, the Mahalanobis Distance could help identify unusual transactions by considering the correlation between different transaction features (like transaction amount, frequency, time of day, etc.). If a transaction's Mahalanobis distance is significantly higher than the typical values, it may indicate fraudulent activity.

Example

```
import numpy as np
import pandas as pd
from scipy.spatial import distance

# Sample data: [Transaction amount, Frequency, Time of day (in hours)]
data = {
    'Transaction_Amount': [50, 75, 100, 200, 5000, 150, 300, 1000, 1200, 60],
    'Transaction_Frequency': [1, 2, 3, 4, 1, 3, 2, 1, 5, 3],
    'Time_of_Day': [10, 14, 15, 16, 3, 12, 9, 18, 7, 11]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Step 1: Calculate the mean and covariance matrix
mean = np.mean(df, axis=0)
covariance_matrix = np.cov(df.T)

# Step 2: Calculate the Mahalanobis Distance for each transaction
inv_cov_matrix = np.linalg.inv(covariance_matrix)  # Inverse of covariance matrix
mahalanobis_distances = []

for i in range(df.shape[0]):
    diff = df.iloc[i] - mean
    mahalanobis_dist = np.sqrt(np.dot(np.dot(diff.T, inv_cov_matrix), diff))
    mahalanobis_distances.append(mahalanobis_dist)

df['Mahalanobis_Distance'] = mahalanobis_distances
```

```
# Step 3: Define a threshold for outlier detection (e.g., 3 standard deviations)
threshold = np.mean(mahalanobis_distances) + 3 * np.std(mahalanobis_distances)

# Step 4: Identify outliers (fraudulent transactions)
df['Is_Fraud'] = df['Mahalanobis_Distance'] > threshold

# Display results
print(df)

# Output outliers (potential fraudulent transactions)
fraudulent_transactions = df[df['Is_Fraud'] == True]
print("\nPotential Fraudulent Transactions:")
print(fraudulent_transactions)
```

Example: Detecting Anomalies using Mahalanobis Distance

```
from scipy.spatial.distance import mahalanobis
import numpy as np

# Sample dataset
X = np.array([[2, 3], [3, 4], [5, 7], [100, 200]])  # The last point is an outlier

# Compute mean and covariance
mean = np.mean(X, axis=0)
cov = np.cov(X.T)

# Compute Mahalanobis distance of each point
distances = [mahalanobis(x, mean, np.linalg.inv(cov)) for x in X]
print("Mahalanobis Distances:", distances)
```

6. Computer Vision and Image Processing

Computer Vision and Image Processing are fields in artificial intelligence that allow computers to interpret and analyze visual data, much like how humans do. The foundation of this process lies in **linear algebra**, as images are represented as matrices or tensors, and various matrix operations are used to extract meaningful information or apply transformations.

Images as Matrices and Tensors

In grayscale images, each pixel value corresponds to an intensity level and is represented as a number. These pixel values form a **matrix** where each entry corresponds to a pixel. For instance, a 256x256 grayscale image will be represented as a 256x256 matrix, where each element contains an intensity value between 0 (black) and 255 (white).

For **color images**, like RGB images, each pixel has three values representing Red, Green, and Blue intensities. These values form a **tensor**,a multi-dimensional array. An RGB image with a size of 256x256 will be represented as a 256x256x3 tensor (3 for the RGB channels).

Example: Convolution for Edge Detection

In edge detection, a common filter is the **Sobel filter**, which is used to highlight the edges in an image by emphasizing areas with high intensity gradients.

Here's an example of how a convolution operation might look in Python using the Sobel operator for edge detection:

```
import numpy as np
import cv2
from scipy.signal import convolve2d
import matplotlib.pyplot as plt

# Load a grayscale image
```

```
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Sobel filter for edge detection
sobel_filter = np.array([[1, 0, -1],
                         [2, 0, -2],
                         [1, 0, -1]])

# Convolve the image with the Sobel filter
edges = convolve2d(image, sobel_filter, mode='same', boundary='wrap')

# Show the original and edge-detected images
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 2, 2)
plt.imshow(edges, cmap='gray')
plt.title('Edge Detected Image')

plt.show()
```

Example: Grayscale Image as a Matrix

```
import cv2
import numpy as np

# Load a grayscale image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Convert to a NumPy matrix
image_matrix = np.array(image)
print("Image Matrix Shape:", image_matrix.shape)
```

QUESTIONS

1. Define a pair of vectors and calculate their dot product using NumPy codes.
2. Given a matrix M = np.array([[2, 4], [3, 6]]), compute its determinant.
3. Do singular value decomposition of a 3x3 matrix that you have selected.
4. Discuss the role of eigenvectors and eigenvalues in implementing Principal Component Analysis.

MODULE 10

ADVANCED NUMPY: ARRAYS AND VECTORIZED COMPUTATION

This module will help you maximize **NumPy arrays** and discover **vectorized computation**. Working with an entire array is very quick with a single line of code using NumPy. Focus on being efficient, rather than trying to do a lot of work! You are about to work efficiently with data and make the calculations in your code more effective and organized.

The reader will have the ability to perform the following tasks:

Understand NumPy Arrays

- State what the difference is between Python lists and NumPy arrays.
- Manage and modify NumPy arrays with ease.

Index, Slice, and Reshape Arrays

- Make use of indexing and slicing to extract items from an array.
- Take an array of any dimension and shape it to the form you need.

Utilize Vectorized Computation

- Carry out mathematical and statistical tasks using vectorized functions.
- Speed up computations using the functions from NumPy.

Work with Multidimensional Arrays

- Construct and control arrays of different dimensions.
- Manipulate matrices with the help of NumPy.

Apply Advanced NumPy Functions

- Use functions like np.where, np.unique, and np.concatenate.
- Perform aggregation and reduction in the system.

INTRODCUTION

NumPy is an important Python library used for doing scientific computations and analyzing data. It allows you to handle large and multi-dimensional arrays and matrices, as well as provides a collection of math functions you can use on them. Because both Pandas, SciPy and Scikit-learn are built on NumPy, it is a key resource for many data scientists and analysts. The ndarray data structure is extremely efficient, making it fast to perform computations with big data.

The features included in NumPy are so useful that it is necessary in the field of data science. First, you can use fast array operations, enabling element-wise operations, broadcasting and vectorized loops which are much speedier than simple Python loops. Secondly, NumPy makes it simpler to work with linear algebra, generate random numbers and use Fourier transforms for applications in statistics, machine learning and signal processing. Moreover, it is compatible with other libraries, allowing data scientists to manage, study and see their data in an efficient manner. Additionally, a NumPy array can be quickly turned into a Pandas DataFrame or passed on to Scikit-learn models.

NumPy is an important tool in data science routines. Because of its efficiency with large data, it is suited for cleaning, changing and adjusting data for tasks like data preprocessing. When using machine learning, NumPy arrays are useful for managing data, carrying out matrix operations and writing algorithms by hand. Furthermore, you can use NumPy's math functions for several

statistical tasks like calculating the mean, variance and correlation. Many tasks in data science would take more time and be much harder without NumPy, as it ensures quick, efficient number computation in Python. All in all, NumPy is an essential part of data science that helps scientists work efficiently and effectively on large amounts of data.

Among NumPy's key features, one is that it is effective when dealing with large amounts of data. This happens for many different reasons:

• NumPy does not mix its data with other built-in Python objects; instead, it stores it all together in one block of memory. By being coded in the C language, these NumPy algorithms can run over the data in memory without any additional checks. NumPy arrays take up less memory than sequences in Python.

• NumPy provides methods to effectively handle many operations on all the data in an array, so loops in Python are not required.

Here, let's compare one million integers in a NumPy array with the same information stored in a Python list:

```
import numpy as np
# Create a NumPy array with 1,000,000 elements
my_arr = np.arange(1000000)

# Create a Python list with 1,000,000 elements
my_list = list(range(1000000))

# Multiply each element in the NumPy array by 2 and measure execution time
%time for _ in range(10): my_arr2 = my_arr * 2

# Multiply each element in the Python list by 2 using list comprehension and measure execution time
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

Table 10-1: Execution Time Results

Operation	**CPU Time**	**Wall Time**
NumPy Array (my_arr * 2)	20 ms (User) + 50 ms (System) = **70 ms**	**72.4 ms**
Python List ([x * 2 for x in my_list])	760 ms (User) + 290 ms (System) = **1.05 s**	**1.05 s**

- ***NumPy is significantly faster*** *than Python lists for large-scale computations.*
- *NumPy performs element-wise operations in* ***optimized C code****, avoiding the overhead of Python loops.*
- *Working with vector operations is more efficient than creating those sequences with list comprehension..*

For this reason, NumPy is used for numerical computations in both data science and machine learning. Using NumPy, the algorithms are at least 10 times faster than pure Python and also take up less memory.

A MULTIDIMENSIONAL ARRAY OBJECT

NumPy's ndarray is a useful tool for working with big datasets in Python. Imagine trying to juggle huge amounts of data,ndarray is your extra pair of hands that lets you handle that data efficiently. It's fast, it's flexible, and it makes number crunching a breeze. With NumPy arrays, you can perform calculations on entire blocks of data, just like you would on individual numbers (scalars) in Python. So, instead of writing loops to go through each item,

NumPy allows you to perform operations on the whole array at once,talk about leveling up your coding game! Now, let's get our hands dirty. To give you a taste of the power, I'm going to import NumPy and generate a small array of random data. Hold on to your seat, because you'll see how smooth the magic happens!

```
import numpy as np
# Generate some random data
data = np.random.randn(2, 3)
print(data)
```

This generates a 2x3 array of random numbers, and you might get an output like this:

```
array([[-0.8013,  1.2496, -0.4081],
       [ 0.2832, -0.3874,  1.7534]])
```

Now, let's perform some operations:

```
# Multiply each element by 10
print(data * 10)
```

This will give you:

```
array([[-8.0131, 12.4959, -4.0811],
       [ 2.8320, -3.8741, 17.5342]])
```

Here, all values in the array are scaled by a factor of 10. Next, let's add the array to itself:

```
# Add the array to itself
print(data + data)
```

The result will be:

```
array([[-1.6026,  2.4992, -0.8162],
       [ 0.5664, -0.7748,  3.5068]])
```

Notice how the operations are applied to the whole array in one go, making your code cleaner and more efficient. This is how magnificent NumPy is!

In this mlodule and throughout the book, I use the standard NumPy convention of always using import numpy as np. You are, of course, welcome to put from numpy import * in your code to avoid having to write np., but I advise against making a habit of

this. The numpy namespace is large and contains a number of functions whose names conflict with built-in Python functions (like min and max).

NDARRAY

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the *data type* of the array:

```
In [6]: data.shape
Out[6]: (2, 3)
In [7]: data.dtype
Out[7]: dtype('float64')
```

This module will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it's not necessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.

Whenever you see "array," "NumPy array," or "ndarray" in the text,with few exceptions they all refer to the same thing: the ndarray object.

Creating ndarrays

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [8]: data1 = [6, 7.5, 8, 0, 1]
In [9]: arr1 = np.array(data1)
In [10]: arr1
Out[10]: array([ 6. , 7.5, 8. , 0. , 1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [12]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
In [13]: arr2 = np.array(data2)
In [14]: arr2
Out[14]: array([[1, 2, 3, 4],
[5, 6, 7, 8]])
```

Since data2 was a list of lists, the NumPy array arr2 has two dimensions with shape inferred from the data. We can confirm this by inspecting the ndim and shape attributes:

```
In [15]: arr2.ndim
Out[15]: 2
In [16]: arr2.shape
Out[16]: (2, 4)
```

Unless explicitly specified (more on this later), np.array tries to infer a good data type for the array that it creates. The data type is stored in a special dtype metadata object; for example, in the previous two examples we have:

```
In [17]: arr1.dtype
Out[17]: dtype('float64')
In [18]: arr2.dtype
Out[18]: dtype('int64')
```

In addition to np.array, there are a number of other functions for creating new arrays. As examples, zeros and ones create arrays of 0s or 1s, respectively, with a given length or shape empty creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [19]: np.zeros(10)
Out[19]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
In [20]: np.zeros((3, 6))
Out[20]:
array([[ 0., 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0., 0.],
```

```
[ 0., 0., 0., 0., 0., 0.]])
```

```
In [21]: np.empty((2, 3, 2))
```

```
Out[21]:
array([[[ 0., 0.],
[ 0., 0.],
[ 0., 0.]],
[[ 0., 0.],
[ 0., 0.],
[ 0., 0.]]])
```

It's not safe to assume that np.empty will return an array of all zeros. In some cases, it may return uninitialized "garbage" values. arange is an array-valued version of the built-in Python range function:

```
In [22]: np.arange(15)
```

```
Out[22]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

See Table 10-2 for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be float64 (floating point).

Table 10-2: Array Creation Functions for NumPy:

Function	Description
np.array()	Creates an array from a Python list or iterable.
np.zeros()	Creates an array filled with zeros.
np.ones()	Creates an array filled with ones.
np.full()	Creates an array filled with a specified value.
np.eye()	Creates an identity matrix (diagonal elements = 1).
np.arange()	Creates an array with a sequence of numbers (like range()).
np.linspace()	Creates an array with evenly spaced values between a start and end.

Function	Description
np.random.rand()	Generates an array of random values from a uniform distribution.
np.random.randn()	Generates an array of random values from a normal distribution.
np.random.randint()	Generates an array of random integers within a range.
np.empty()	Creates an uninitialized array (values may be arbitrary).

This table provides a quick reference for **array creation functions in NumPy**, essential for efficient numerical computing.

Data Types (dtype) for NumPy ndarrays

In NumPy, every ndarray has an associated **data type (dtype)** that determines how the elements are stored in memory. This allows NumPy to optimize performance and memory usage.

Table 10-3: Common NumPy Data Types

Data Type	Description	Example
int8, int16, int32, int64	Integer types with varying bit sizes	np.array([1, 2, 3], dtype=np.int32)
uint8, uint16, uint32, uint64	Unsigned integer types (only positive values)	np.array([255, 128, 64], dtype=np.uint8)
float16, float32, float64 (or float)	Floating-point numbers with different precision levels	np.array([1.5, 2.3, 3.7], dtype=np.float64)
complex64, complex128	Complex numbers with real and imaginary parts	np.array([1+2j, 3+4j], dtype=np.complex128)
Bool	Boolean values (True	np.array([True, False, True],

Data Type	Description	Example
	or False)	dtype=np.bool_)
str or object	String or object data types	np.array(["hello", "world"], dtype=np.str_)

Checking and Changing Data Types

Check data type:

- arr = np.array([1, 2, 3], dtype=np.int32)
- print(arr.dtype) # Output: int32

Change data type (astype):

- arr_float = arr.astype(np.float64)
- print(arr_float.dtype) # Output: float64

- *Choosing the right dtype helps optimize **performance** and **memory usage**.*
- *Use **smaller data types** (int8, float16) when memory is a constraint.*
- *Use astype() for **type conversion** when necessary.*

ARITHMETIC WITH NUMPY ARRAYS

Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this *vectorization*. Any arithmetic operations between equal-size arrays apply the operation element-wise:

```
import numpy as np
# Creating a 2D NumPy array
arr = np.array([[1., 2., 3.],
                [4., 5., 6.]])

# Display the array
```

```
print("Original Array:")
print(arr)

# Element-wise multiplication
arr_mult = arr * arr
print("\nElement-wise Multiplication:")
print(arr_mult)

# Element-wise subtraction
arr_sub = arr - arr
print("\nElement-wise Subtraction:")
print(arr_sub)
```

- ***arr * arr*** *→ Performs element-wise multiplication.*
- ***arr - arr*** *→ Performs element-wise subtraction, resulting in an array of zeros.*

Scalar arithmetic operations are applied element-wise to every value in the array.

For example:

```
1 / arr
```

Produces:

```
array([[ 1. , 0.5 , 0.3333],
       [ 0.25 , 0.2 , 0.1667]])
```

Similarly, performing exponentiation applies the operation to each element:

```
arr ** 0.5
```

Results in:

```
array([[ 1. , 1.4142, 1.7321],
       [ 2. , 2.2361, 2.4495]])
```

When comparing arrays of equal size, the operation is performed element-wise, generating a boolean array:

```
arr2 = np.array([[0., 4., 1.],
                 [7., 2., 12.]])
arr2 > arr
```

Produces:

```
array([[False, True, False],
       [ True, False, True]], dtype=bool)
```

For arrays with different dimensions, operations follow **broadcasting rules**, which will be covered in more detail in Appendix A. However, a deep understanding of broadcasting is not essential for most of this book.

INDEXING, SLICING, AND RESHAPING ARRAYS

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

Example 1

```
In [30]: arr = np.arange(10)
In [31]: arr
Out[31]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [32]: arr[5]
```

```
Out[32]: 5
```

```
In [33]: arr[5:8]
```

```
Out[33]: array([5, 6, 7])
```

```
In [34]: arr[5:8] = 12
In [35]: arr
```

```
Out[35]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

Example 2

```
arr = np.array([10, 20, 30, 40, 50])

# Accessing elements
print(arr[0])  # First element
```

```
print(arr[-1]) # Last element

# Slicing
print(arr[1:4])  # Elements from index 1 to 3
```

Reshaping Arrays

In **NumPy**, reshaping is the process of changing the shape of an existing array without modifying its data. This allows you to transform an array into a different configuration (e.g., from a flat 1D array to a 2D matrix or from a 2D matrix to a higher-dimensional tensor) while keeping the data intact. It's like rearranging the seats at a party without changing the number of guests!

For example, if you have a 1D array with 12 elements, you can reshape it into a 2D array with 3 rows and 4 columns. The total number of elements before and after reshaping must remain the same, but you can adjust how they are arranged.

```
import numpy as np
# Create a 1D array of 12 elements
data = np.arange(12)
print("Original Array:", data)

# Reshape it into a 2D array (3 rows, 4 columns)
reshaped_data = data.reshape(3, 4)
print("Reshaped Array:\n", reshaped_data)
```

As you can see in the first example, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are *views* on the original array.

This means that the data is not copied, and any modifications to the view will be reflected in the source array.

To give an example of this, I first create a slice of arr:

```
In [36]: arr_slice = arr[5:8]
In [37]: arr_slice
```

```
Out[37]: array([12, 12, 12])
```

Now, when I change values in arr_slice, the mutations are reflected in the original array arr:

```
In [38]: arr_slice[1] = 12345
In [39]: arr
```

```
Out[39]: array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8,
9])
```

The "bare" slice [:] will assign to all values in an array:

```
In [40]: arr_slice[:] = 64
In [41]: arr
```

```
Out[41]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages that copy data more eagerly. As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array,for example,

```
arr[5:8].copy().
```

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [42]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In [43]: arr2d[2]
```

```
Out[43]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements.

So these are equivalent:

```
In [44]: arr2d[0][2]
Out[44]: 3
In [45]: arr2d[0, 2]
Out[45]: 3
```

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array arr3d:

```
In [46]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
In [47]: arr3d
```

```
Out[47]:
array([[[ 1, 2, 3],
[ 4, 5, 6]],
[[ 7, 8, 9],
[10, 11, 12]]])
```

arr3d[0] is a 2×3 array:

```
In [48]: arr3d[0]
```

```
Out[48]:
array([[1, 2, 3],
[4, 5, 6]])
```

Both scalar values and arrays can be assigned to arr3d[0]:

```
In [49]: old_values = arr3d[0].copy()
In [50]: arr3d[0] = 42
In [51]: arr3d
```

```
Out[51]:
array([[[42, 42, 42],
[42, 42, 42]],
[[ 7, 8, 9],
[10, 11, 12]]])
```

```
In [52]: arr3d[0] = old_values
In [53]: arr3d
```

```
Out[53]:
array([[[ 1, 2, 3],
[ 4, 5, 6]],
[[ 7, 8, 9],
[10, 11, 12]]])
```

Similarly, arr3d[1, 0] gives you all of the values whose indices start with (1, 0), forming a 1 dimensional array:

```
In [54]: arr3d[1, 0]
Out[54]: array([7, 8, 9])
```

This expression is the same as though we had indexed in two steps:

```
In [55]: x = arr3d[1]
In [56]: x
Out[56]:
array([[ 7, 8, 9],
[10, 11, 12]])
In [57]: x[0]
Out[57]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax:

```
In [58]: arr
Out[58]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
In [59]: arr[1:6]
Out[59]: array([ 1, 2, 3, 4, 64])
```

Consider the two-dimensional array from before, arr2d. Slicing this array is a bit different:

```
In [60]: arr2d
Out[60]:
array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
In [61]: arr2d[:2]
Out[61]:
array([[1, 2, 3],
[4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression arr2d[:2] as "select the first two rows of arr2d."

You can pass multiple slices just like you can pass multiple indexes:

```
In [62]: arr2d[:2, 1:]
Out[62]:
array([[2, 3],
[5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices. For example, I can select the second row but only the first two columns like so:

```
In [63]: arr2d[1, :2]
Out[63]: array([4, 5])
```

Similarly, I can select the third column but only the first two rows like so:

```
In [64]: arr2d[:2, 2]
Out[64]: array([3, 6])
```

Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [65]: arr2d[:, :1]
Out[65]:
array([[1],
[4],
[7]])
```

Of course, assigning to a slice expression assigns to the whole selection:

```
In [66]: arr2d[:2, 1:] = 0
In [67]: arr2d
Out[67]:
array([[1, 0, 0],
[4, 0, 0],
```

```
[7, 8, 9]])
```

Boolean Indexing

Let's consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the randn function in numpy.random to generate some random normally distributed data:

```
In [68]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
In [69]: data = np.random.randn(7, 4)
In [70]: names
```

```
Out[70]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
dtype='<U4')
```

```
In [71]: data
```

```
Out[71]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.0072, -1.2962, 0.275 , 0.2289],
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 1.669 , -0.4386, -0.5397, 0.477 ],
[ 3.2489, -1.0212, -0.5771, 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[-0.7135, -0.8312, -2.3702, -1.8608]])
```

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [72]: names == 'Bob'
Out[72]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [73]: data[names == 'Bob']
Out[73]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.669 , -0.4386, -0.5397, 0.477 ]])
```

The boolean array must be of the same length as the array axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers; more on this later).

Boolean selection will not fail if the boolean array is not the correct length, so I recommend care when using this feature.

In these examples, I select from the rows where names == 'Bob' and index the columns, too:

```
In [74]: data[names == 'Bob', 2:]
Out[74]:
array([[ 0.769 , 1.2464],
[-0.5397, 0.477 ]])
In [75]: data[names == 'Bob', 3]
Out[75]: array([ 1.2464, 0.477 ])
```

To select everything but 'Bob', you can either use != or negate the condition using ~:

```
In [76]: names != 'Bob'
Out[76]: array([False, True, True, False, True, True, True], dtype=bool)
In [77]: data[~(names == 'Bob')]
Out[77]:
array([[ 1.0072, -1.2962, 0.275 , 0.2289],
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 3.2489, -1.0212, -0.5771, 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[-0.7135, -0.8312, -2.3702, -1.8608]])
```

The ~ operator can be useful when you want to invert a general condition:

```
In [78]: cond = names == 'Bob'
In [79]: data[~cond]
Out[79]:
array([[ 1.0072, -1.2962, 0.275 , 0.2289],
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 3.2489, -1.0212, -0.5771, 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[-0.7135, -0.8312, -2.3702, -1.8608]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or):

```
In [80]: mask = (names == 'Bob') | (names == 'Will')
In [81]: mask
Out[81]: array([ True, False, True, True, True, False, False], dtype=bool)
In [82]: data[mask]
Out[82]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 1.669 , -0.4386, -0.5397, 0.477 ],
[ 3.2489, -1.0212, -0.5771, 0.1241]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.

The Python keywords AND and OR do not work with boolean arrays. Use & (AND) and | (OR) instead. Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do:

```
In [83]: data[data < 0] = 0
In [84]: data
Out[84]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.0072, 0. , 0.275 , 0.2289],
[ 1.3529, 0.8864, 0. , 0. ],
[ 1.669 , 0. , 0. , 0.477 ],
[ 3.2489, 0. , 0. , 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[ 0. , 0. , 0. , 0. ]])
```

Setting whole rows or columns using a one-dimensional boolean array is also easy:

```
In [85]: data[names != 'Joe'] = 7
In [86]: data
Out[86]:
array([[ 7. , 7. , 7. , 7. ],
[ 1.0072, 0. , 0.275 , 0.2289],
```

```
[ 7. , 7. , 7. , 7. ],
[ 7. , 7. , 7. , 7. ],
[ 7. , 7. , 7. , 7. ],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[ 0. , 0. , 0. , 0. ]])
```

As we will see later, these types of operations on two-dimensional data are convenient to do with pandas

TRANSPOSING ARRAYS AND SWAPPING AXES

Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and also the special T attribute:

```
In [126]: arr = np.arange(15).reshape((3, 5))
In [127]: arr
Out[127]:
array([[ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]])
In [128]: arr.T
Out[128]:
array([[ 0, 5, 10],
[ 1, 6, 11],
[ 2, 7, 12],
[ 3, 8, 13],
[ 4, 9, 14]])
```

When doing matrix computations, you may do this very often,for example, when computing the inner matrix product using np.dot:

```
In [129]: arr = np.random.randn(6, 3)
In [130]: arr
Out[130]:
array([[-0.8608, 0.5601, -1.2659],
[ 0.1198, -1.0635, 0.3329],
[-2.3594, -0.1995, -1.542 ],
[-0.9707, -1.307 , 0.2863],
[ 0.378 , -0.7539, 0.3313],
```

```
[ 1.3497, 0.0699, 0.2467]])
In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291, 0.9394, 4.948 ],
[ 0.9394, 3.7662, -1.3622],
[ 4.948 , -1.3622, 4.3437]])
```

For higher dimensional arrays, transpose will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))
In [133]: arr
Out[133]:
array([[[ 0, 1, 2, 3],
[ 4, 5, 6, 7]],
[[ 8, 9, 10, 11],
[12, 13, 14, 15]]])
In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0, 1, 2, 3],
[ 8, 9, 10, 11]],
[[ 4, 5, 6, 7],
[12, 13, 14, 15]]])
```

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.

Simple transposing with .T is a special case of swapping axes. ndarray has the method swap axes, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In [135]: arr
Out[135]:
array([[[ 0, 1, 2, 3],
[ 4, 5, 6, 7]],
[[ 8, 9, 10, 11],
[12, 13, 14, 15]]])
In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0, 4],
[ 1, 5],
```

```
[ 2, 6],
[ 3, 7]],
[[ 8, 12],
[ 9, 13],
[10, 14],
[11, 15]]])
```

swapaxes similarly returns a view on the data without making a copy

MATHEMATICAL AND STATISTICAL METHODS

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (often called *reductions*) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function.

Here I generate some normally distributed random data and compute some aggregate statistics:

```
In [177]: arr = np.random.randn(5, 4)
In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149, 2.0037, 0.0296],
[ 0.7953, 0.1181, -0.7485, 0.585 ],
[ 0.1527, -1.5657, -0.5625, -0.0327],
[-0.929 , -0.4826, -0.0363, 1.0954],
[ 0.9809, -0.5895, 1.5817, -0.5287]])
In [179]: arr.mean()
Out[179]: 0.19607051119998253
In [180]: np.mean(arr)
Out[180]: 0.19607051119998253
In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

Functions like mean and sum take an optional axis argument that computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 , 0.1875, -0.502 , -0.0881, 0.3611])
In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345, 2.2381, 1.1486])
```

Here, arr.mean(1) means "compute mean across the columns" where arr.sum(0) means "compute sum down the rows."

Other methods like cumsum and cumprod do not aggregate, instead producing an array of the intermediate results:

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
In [185]: arr.cumsum()
Out[185]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

In multidimensional arrays, accumulation functions like cumsum return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower dimensional slice:

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
In [187]: arr
Out[187]:
array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0, 1, 2],
[ 3, 5, 7],
[ 9, 12, 15]])
In [189]: arr.cumprod(axis=1)
Out[189]:
array([[ 0, 0, 0],
[ 3, 12, 60],
[ 6, 42, 336]])
```

Table 10-4: Basic Array Statistical Methods

Method	Description
Sum	Computes the sum of all elements in the array or along a specified axis. Zero-length arrays return 0.
Mean	Calculates the arithmetic mean. Zero-length arrays return NaN.
std, var	Computes the standard deviation and variance, respectively, with an optional degrees of freedom adjustment (default denominator is *n*).
min, max	Returns the minimum and maximum values in the array.
argmin, argmax	Returns the indices of the minimum and maximum elements, respectively.
cumsum	Computes the cumulative sum of elements, starting from 0.
cumprod	Computes the cumulative product of elements, starting from 1.

METHODS FOR BOOLEAN ARRAYS

Boolean values are coerced to 1 (True) and 0 (False) in the preceding methods. Thus, sum is often used as a means of counting True values in a boolean array:

```
In [190]: arr = np.random.randn(100)
In [191]: (arr > 0).sum() # Number of positive values
```

```
Out[191]: 42
```

There are two additional methods, any and all, useful especially for boolean arrays. any tests whether one or more values in an array is True, while all checks if every value is True:

```
In [192]: bools = np.array([False, False, True, False])
In [193]: bools.any()
```

```
Out[193]: True
In [194]: bools.all()
Out[194]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

SORTING NUMPY ARRAY

Like Python's built-in list type, NumPy arrays can be sorted in-place with the sort method:

```
In [195]: arr = np.random.randn(6)
In [196]: arr
Out[196]: array([ 0.6095, -0.4938, 1.24 , -0.1357, 1.43 , -0.8469])
In [197]: arr.sort()
In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357, 0.6095, 1.24 , 1.43 ])
```

You can sort each one-dimensional section of values in a multidimensional array inplace along an axis by passing the axis number to sort:

```
In [199]: arr = np.random.randn(5, 3)
In [200]: arr
Out[200]:
array([[ 0.6033, 1.2636, -0.2555],
[-0.4457, 0.4684, -0.9616],
[-1.8245, 0.6254, 1.0229],
[ 1.1074, 0.0909, -0.3501],
[ 0.218 , -0.8948, -1.7415]])
In [201]: arr.sort(1)
In [202]: arr
Out[202]:
array([[-0.2555, 0.6033, 1.2636],
[-0.9616, -0.4457, 0.4684],
[-1.8245, 0.6254, 1.0229],
[-0.3501, 0.0909, 1.1074],
[-1.7415, -0.8948, 0.218 ]])
```

The top-level method np.sort returns a sorted copy of an array instead of modifying the array in-place. A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [203]: large_arr = np.random.randn(1000)
In [204]: large_arr.sort()
In [205]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[205]: -1.5311513550102103
```

UNIQUE AND OTHER SET LOGIC

NumPy has some basic set operations for one-dimensional ndarrays. A commonly used one is np.unique, which returns the sorted unique values in an array:

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
dtype='<U4')
In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

Contrast np.unique with the pure Python alternative:

```
In [210]: sorted(set(names))
Out[210]: ['Bob', 'Joe', 'Will']
```

Another function, np.in1d, tests membership of the values in one array in another, returning a boolean array:

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])
In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False, True, True, False, True], dtype=bool)
```

Table 10-5: Array Set Operations

Method	Description
unique(x)	Computes the sorted, unique elements in *x*.

Method	Description
intersect1d(x, y)	Finds the sorted, common elements present in both *x* and *y*.
union1d(x, y)	Computes the sorted union of elements from *x* and *y*.
in1d(x, y)	Returns a boolean array indicating whether each element of *x* is present in *y*.
setdiff1d(x, y)	Computes the set difference, returning elements in *x* that are not in *y*.
setxor1d(x, y)	Finds the symmetric difference,elements that appear in either *x* or *y*, but not in both.

FILE INPUT AND OUTPUT WITH ARRAYS

NumPy is able to save and load data to and from disk either in text or binary format. In this section I only discuss NumPy's built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data.

np.save and np.load are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension *.npy*:

```
In [213]: arr = np.arange(10)
In [214]: np.save('some_array', arr)
```

If the file path does not already end in *.npy*, the extension will be appended. The array on disk can then be loaded with np.load:

```
In [215]: np.load('some_array.npy')
```

```
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in an uncompressed archive using np.savez and passing the arrays as keyword arguments:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an *.npz* file, you get back a dict-like object that loads the individual arrays lazily:

```
In [217]: arch = np.load('array_archive.npz')
In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If your data compresses well, you may wish to use numpy.savez_compressed instead:

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

PSEUDORANDOM NUMBER GENERATION

The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using normal:

```
In [238]: samples = np.random.normal(size=(4, 4))
In [239]: samples
Out[239]:
array([[ 0.5732, 0.1933, 0.4429, 1.2796],
[ 0.575 , 0.4339, -0.7658, -1.237 ],
[-0.5367, 1.8545, -0.92 , -0.1082],
[ 0.1525, 0.9435, -1.0953, -0.144 ]])
```

Python's built-in random module, by contrast, only samples one value at a time. As you can see from this benchmark, numpy.random is well over an order of magnitude faster for generating very large samples:

```
In [240]: from random import normalvariate
In [241]: N = 1000000
In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

We say that these are *pseudorandom* numbers because they are generated by an algo-

rithm with deterministic behavior based on the *seed* of the random number generator. You can change NumPy's random number generation seed using np.random.seed:

```
In [244]: np.random.seed(1234)
```

The data generation functions in numpy.random use a global random seed. To avoid global state, you can use numpy.random.RandomState to create a random number generator isolated from others:

```
In [245]: rng = np.random.RandomState(1234)
In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 , 1.4327, -0.3127, -0.7206, 0.8872, 0.8596,
-0.6365, 0.0157, -2.2427])
```

Table 10-6: Partial List of numpy.random Functions

Function	Description
Seed	Sets the seed for the random number generator to ensure it can be repeated later.
permutation	Returns a sequence of numbers arranged randomly or by a given range in a random order.
Shuffle	Randomly permutes a sequence in-place.
Rand	Generates samples from a uniform distribution over [0, 1).
Randint	Draws random integers from a specified low-to-high range.
Randn	Generates samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface).
Binomial	The process involves obtaining samples from a binomial distribution.

Function	Description
Normal	Creates samples using a normal (Gaussian) distribution.
Beta	Technique for drawing samples from beta distribution.
Chisquare	Based on the selected method, will sample from a chi-square distribution.
Gamma	Gives samples from a gamma distribution.
uniform	Makes samples using the uniform distribution in the range [0, 1).

ADVANCED NUMPY FUNCTIONS

Using np.where, np.unique, and np.concatenate in NumPy

With NumPy, you can easily and efficiently handle and work with arrays. A lot of people use np.where, np.unique, and np.concatenate functions regularly. We use these functions for tasks such as filtering under conditions, identifying different elements and combining arrays. Each function is explained in detail here with some examples.

1. np.where()

It is used to carry out actions on elements or filter data according to given conditions using np.where(). It provides the indices or values that meet the given criterion.

Syntax:

np.where(condition, [x, y])

- This returns values from x where True and values from y where False.
- If only the condition is given, it produces a list of where the condition is True.

Examples:

1. **Find indices where elements satisfy a condition:**

```
import numpy as np
array = np.array([1, 2, 3, 4, 5])
indices = np.where(array > 3)  # Output: (array([3, 4]),)
print(indices)
```

2. **Replace elements based on a condition:**

```
array = np.array([1, 2, 3, 4, 5])
result = np.where(array > 3, 10, 0)  # Replace elements > 3 with 10, e
lse 0
print(result)  # Output: [0, 0, 0, 10, 10]
```

2. np.unique()

The np.unique() function is used to **find unique elements** in an array. It cleans up the array and returns a list of all the different elements, sorted in order.

Syntax:

np.unique(array, return_index=False, return_counts=False, axis=0)

- return_index: If True, the function will return the indices of where each unique element first appears..
- return_counts: If True, returns the count of each unique element.
- axis: Specifies the axis along which to find unique elements (for multi-dimensional arrays).

Examples:

1. **Find unique elements:**

```
array = np.array([1, 2, 2, 3, 4, 4, 5])
unique_elements = np.unique(array)
print(unique_elements)  # Output: [1, 2, 3, 4, 5]
```

2. **Find unique elements with counts:**

```
array = np.array([1, 2, 2, 3, 4, 4, 5])
unique_elements, counts = np.unique(array, return_counts=True)
print(unique_elements)  # Output: [1, 2, 3, 4, 5]
print(counts)  # Output: [1, 2, 1, 2, 1]
```

3. **Find unique rows in a 2D array**:

```
array = np.array([[1, 2], [3, 4], [1, 2]])
unique_rows = np.unique(array, axis=0)
print(unique_rows)  # Output: [[1, 2], [3, 4]]
```

3. np.concatenate()

The np.concatenate() function is used to **combine two or more arrays** along a specified axis. It can be helpful for integrating different sets of data or adding more rows/columns to an array.

Syntax:

```
np.concatenate((array1, array2, ...), axis=0)
```

- axis=0: Concatenates vertically (row-wise).
- axis=1: Concatenates horizontally (column-wise).

Examples:

1. **Concatenate two 1D arrays**:

```
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
result = np.concatenate((array1, array2))
print(result)  # Output: [1, 2, 3, 4, 5, 6]
```

2. **Concatenate two 2D arrays vertically**:

```
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])
result = np.concatenate((array1, array2), axis=0)
print(result)  # Output: [[1, 2], [3, 4], [5, 6], [7, 8]]
```

3. **Concatenate two 2D arrays horizontally**:

```
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])
result = np.concatenate((array1, array2), axis=1)
print(result)  # Output: [[1, 2, 5, 6], [3, 4, 7, 8]]
```

- **np.where**: *Useful for filtering data, replacing values, or performing conditional operations.*
- **np.unique**: *helps to identify each distinct value, eliminate similar values or count instances.*
- **np.concatenate**: *Use np.concatenate to link one array to another, merge several arrays or add rows or columns to a data set.*

AGGREGATION AND REDUCTION OPERATIONS IN NUMPY

Aggregation and reduction make it easy to organize and examine data with NumPy. With these operations, you can perform statistics or decrease array size by computing functions like sum, mean, max, min and similar. NumPy includes efficient and fast functions that rely on vectorized computation to perform these operations.

Key Aggregation and Reduction Functions

Here are a few of the usual aggregation and reduction functions found in NumPy:

1. **np.sum()**:
   Calculates the total of all the array values using a specified axis.
   Example:

```
array = np.array([1, 2, 3, 4])
result = np.sum(array)  # Output: 10
```

2. **np.mean()**:
   Calculates the arithmetic mean of array elements.

Example:

```
array = np.array([1, 2, 3, 4])
result = np.mean(array)  # Output: 2.5
```

3. **np.min() and np.max()**:
   Find the minimum and maximum values in an array.
   Example:

```
array = np.array([1, 2, 3, 4])
min_val = np.min(array)  # Output: 1
max_val = np.max(array)  # Output: 4
```

4. **np.prod()**:
   Computes the product of all elements in an array.
   Example:

```
array = np.array([1, 2, 3, 4])
result = np.prod(array)  # Output: 24
```

5. **np.std()**:
   Calculates the standard deviation of array elements.
   Example:

```
array = np.array([1, 2, 3, 4])
result = np.std(array)  # Output: 1.118 (standard deviation)
```

6. **np.var()**:
   Computes the variance of array elements.
   Example:

```
array = np.array([1, 2, 3, 4])
result = np.var(array)  # Output: 1.25 (variance)
```

7. **np.cumsum() and np.cumprod()**:
   Compute the cumulative sum and cumulative product of array elements.
   Example:

```
array = np.array([1, 2, 3, 4])
cumsum = np.cumsum(array)  # Output: [1, 3, 6, 10]
cumprod = np.cumprod(array)  # Output: [1, 2, 6, 24]
```

8. **np.argmin() and np.argmax()**:
   Return the indices of the minimum and maximum values in an array.
   Example:

```
array = np.array([1, 2, 3, 4])
argmin = np.argmin(array)  # Output: 0 (index of min value)
argmax = np.argmax(array)  # Output: 3 (index of max value)
```

Axis Parameter in Aggregation

The operation can be carried out on different axes in NumPy, depending on the value you set for the axis parameter in the aggregation function. For example:

- **axis=0**: Operate along rows (column-wise).
- **axis=1**: Operate along columns (row-wise).

Example:

```
array = np.array([[1, 2, 3], [4, 5, 6]])
row_sum = np.sum(array, axis=1)  # Output: [6, 15] (sum along rows)
col_sum = np.sum(array, axis=0)  # Output: [5, 7, 9] (sum along columns)
```

Importance of aggregation and reduction in Data Science

Aggregation and reduction operations are critical in data science for:

1. **Summarizing Data**: Perform calculations for mean, sum or variance in a fast and effective way.
2. **Data Preprocessing**: Cut down the features in your data or select the main features.
3. **Performance Optimization**: Make use of NumPy's vectorized functions for quicker computations than using Python loops.
4. **Exploratory Data Analysis (EDA)**: Get understanding of the distributions and changes in your data.

Example of the Use Case

For example, you have a 2D array that shows sales figures for several products each month. You have the option of using aggregation functions to:

- Sum the sales for each product by using the command (np.sum(axis=0)).
- Get the average sales for each month (np.mean(axis=1)).
- Identify the month with the highest sales (np.argmax()).

```
sales = np.array([[100, 150, 200], [250, 300, 350], [400, 450, 500]])
total_sales = np.sum(sales, axis=0)  # Total sales per product
avg_monthly_sales = np.mean(sales, axis=1)  # Average sales per month
```

QUESTIONS

1. Create a 3x3 NumPy array and compute its transpose.
2. Generate an array of 10 random numbers and find the mean.
3. Use np.where to label elements in an array as "Even" or "Odd".
4. How do you use np.where() to filter elements in an array?
5. What is the purpose of the np.unique() function?
6. How do you calculate the mean of a NumPy array along a specific axis?
7. What is the difference between np.sum() and np.cumsum()?
8. How do you create a 3x3 identity matrix using NumPy?
9. Create a NumPy array with the values [1, 2, 3, 4, 5] and print its shape and data type.
10. Perform element-wise multiplication on two arrays: a = [1, 2, 3] and b = [4, 5, 6].
11. Reshape a 1D array [1, 2, 3, 4, 5, 6] into a 2x3 matrix.
12. Extract the second column from the following 2D array:
    ```
    array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    ```
13. Use broadcasting to add the scalar value 5 to every element in a 2D array.
14. Find the maximum value in the following array:
    ```
    array = np.array([10, 20, 30, 40, 50])
    ```
15. Concatenate two arrays horizontally:
    ```
    a = np.array([1, 2, 3])
    b = np.array([4, 5, 6])
    ```
16. Calculate the mean of the following array along the rows:
    ```
    array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    ```
17. Create a 3D array of shape (2, 3, 4) filled with zeros.

18. Use np.unique() to find unique values in the array:

```
array = np.array([1, 2, 2, 3, 4, 4, 5])
```

MODULE 11

PROBABILITY AND STATISTICS

Probability and statistics form the backbone of data science, enabling data-driven decision-making, uncertainty modeling, and hypothesis testing. This module introduces key concepts in probability and statistics and their applications in data science using Python.

By the end of this module, you will be able to:

- Understand fundamental probability concepts.
- Compute descriptive statistics and visualize data distributions.
- Perform hypothesis testing and inferential statistics.
- Apply probability and statistical techniques in data science applications.

INTRODUCTION

Probability shows how likely it is for an event to take place. In data science, uncertainty is represented by this technique when making predictions.

In recent years, Probability has played a key role in data science by supporting understanding of uncertainty, prediction and drawing conclusions from gathered data.

Basic Probability Concepts

- **Sample Space (S):**
  The sample space in data science includes all the possible results that can appear in a certain dataset. For example:
  - The possible options in a sample space of customer purchases include all products that may be purchased.

    - A/B testing examines every outcome that can happen from users interacting with two versions of a website.
- **Events (E):**
  An event is a particular result highlighted within the data. For example:
    - An event could be a customer making a purchase above $100.
    - In anomaly detection, an event might be an outlier in a dataset.
- Probability of an Event (P(E)): Ranges from 0 (impossible) to 1 (certain).
- Conditional Probability: The probability of an event given that another event has occurred.
- Bayes' Theorem: Used to change the probability of a reason due to new information.

Example: Computing Basic Probability

```
import random
def simulate_coin_toss(n_trials=10000):
    """
    Simulates a series of coin tosses and calculates the probability of getting heads.
    Args:
        n_trials (int): The number of coin tosses to simulate. Default is 10,000.
    Returns:
        float: The probability of getting heads.
    """
    # Simulate coin tosses: 1 for heads, 0 for tails
    heads = sum(random.choice([0, 1]) for _ in range(n_trials))
    # Calculate the probability of heads
    probability = heads / n_trials
    return probability
def main():
    # Set the number of trials
```

```
    n_trials = 10000
    # Calculate the probability of heads
    probability = simulate_coin_toss(n_trials)
    # Print the result
    print(f"Probability of heads in a fair coin toss (after {n_trials} trials): {probability:.4f}")

# Run the program
if __name__ == "__main__":
    main()
```

Example 1 simulates the probability of getting heads in a fair coin toss by running a large number of trials (n_trials). Here's a breakdown of how it works:

1. **random.choice([0, 1])**: This randomly selects either 0 (representing tails) or 1 (representing heads) with equal probability.
2. **sum(...)**: The sum function adds up all the 1s (heads) from the n_trials coin tosses.
3. **heads / n_trials**: This calculates the proportion of heads in the total number of trials, which approximates the probability of getting heads.

The code is designed to print the chance of heads coming up when probabilities are compared.

Why Probability and Statistics are essential in the world of data science.

1. **Uncertainty Modeling:** To assess and predict outcomes with data, probability helps us measure the randomness and uncertainty it contains.
2. **Data Analysis:** Using Statistics, data can be summarized, shown visually and studied, leading to valuable discoveries based on the information.

3. **Hypothesis Testing:** Statistical tests are used to confirm and draw conclusions about populations by studying a part of them.
4. **Decision-Making:** Using Probability and statistics, data scientists are prepared to decide confidently when there are uncertainties in A/B testing and evaluating machine learning models.

RULES OF PROBABILITY

Addition Rule

- When calculating the probability that either of two events will occur, you rely on the addition rule.:
    - Customer Segmentation: Figuring out the odds that a customer falls under either Segment A or Segment B.
    - Risk Assessment: Calculating the likelihood that one or both of two particular risks may affect a company's process.

  Example:

  For this example, we have a dataset that indicates:
    - There is a 40% chance that a customer will buy Product A.
    - Thirty percent chance that a customer will select Product B.
    - It is likely that a customer will buy both products on only 0.1 occasions.

  The addition rule applied yields the answer:
- $P(A \cup B) = P(A) + P(P(B) - P(A \cap B) = 0.4 + 0.3 - 0.1 = 0.6$

  A customer, then, has a 60% chance of buying either Product A or Product B.

Multiplication Rule

- It allows you to find the probability that two events happen at the same time which is important for many things:
    - Feature Engineering: Determining the combination of age and income (e.g., two features) with an equal likelihood of appearing in a dataset.
    - Bayesian Inference: Updating chances of different outcomes for increased certainty.

  Example:

  Suppose:
    - Only a 0.2 chance exists that a person will click an ad.
    - Half of the customers who click on the ad end up making a purchase.

  Using the multiplication rule we can multiply the exponent by each term in the product:
- $P(\text{Click} \cap \text{Purchase}) = P(\text{Click}) \cdot P(\text{Purchase}|\text{Click}) = 0.2 \cdot 0.5 = 0.1$

  In this case, only 10% of the customers viewing the ad will make a purchase.

RANDOM VARIABLES, PROBABILITY DISTRIBUTIONS, AND CONDITIONAL PROBABILITY

1. RANDOM VARIABLES

A random variable is a variable whose possible values are outcomes of a random phenomenon. It is a fundamental concept in probability and statistics, widely used in data science for modeling and analysis.

Discrete vs. Continuous Random Variables

Discrete Random Variables:

- Take on a countable number of distinct values.
- Examples: Number of customers arriving at a store, number of defective items in a batch.
- Represented by a Probability Mass Function (PMF), which gives the probability of each possible value.

Continuous Random Variables:

- Take on an uncountable number of values within a range.
- Examples: Height of individuals, time between customer arrivals.
- Represented by a Probability Density Function (PDF), which describes the relative likelihood of the variable taking on a specific value.

2. COMMON PROBABILITY DISTRIBUTIONS

Discrete Distributions

Binomial Distribution:

- Models the number of successes in a fixed number of independent trials, each with the same probability of success.
- Example: Number of heads in 10 coin flips.

Binomial Distribution Probality Mass Fucntion:

$$P(X = k) = C(n, k) \cdot p^k \cdot (1 - p)^{n-k}$$

Where:

n = *Number of trials (a fixed number of independent experiments).*

k = *Number of successes (must be an integer* $0 \leq k \leq n$*).*

p = *Probability of success in a single trial (*$0 \leq p \leq 1$*).*

- $C(n,k)$ = **Combination** ("n choose k"), calculated as:

$$C(n,k) = \frac{n!}{k!\,(n-k)!}$$

This counts the number of ways to choose k successes out of n trials.

Example:

If you flip a fair coin ($p = 0.5$) 10 times ($n = 10$), the probability of getting exactly 3 heads ($k = 3$) is:

$$P(X = 3) = C(10,3) \cdot (0.5)^3 \cdot (0.5)^7 = 120 \cdot \frac{1}{1024} \approx 0.117$$

Poisson Distribution:

- Models the number of events occurring in a fixed interval of time or space.
- Example: Number of emails received in an hour.

Poisson Distribution PMF:

$$P(X = k) = \frac{\iota \lambda^k e^{-\lambda}}{k!}$$

Where:

k = Number of events (a non-negative integer, $k = 0,1,2,\ldots$).

λ = Average rate of occurrence (mean number of events in the interval).

e = Euler's number (~ 2.71828).

Example

Suppose a help desk receives an average of $\lambda = 5$ calls per hour. **What is the probability of receiving exactly 3 calls in an hour?**

$$P(X = 3) = \frac{5^3 e^{-5}}{3!} = \frac{125 \cdot e^{-5}}{6} \approx 0.1404\ (14.04\%).$$

Continuous Distributions

Normal (Gaussian) Distribution:

- Symmetric, bell-shaped distribution characterized by its mean (μμ) and standard deviation (σσ).
- Example: Distribution of heights in a population.

Normal Probability Density Function:

$$f(x) = \frac{1}{\sigma\sqrt{2\Pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where:

x = Continuous random variable.

μ = Mean (location parameter, center of the distribution).

σ = Standard deviation (scale parameter, measures spread).

σ^2 = Variance.

$\pi \approx 3.14159$, $e \approx 2.71828$.

Uniform Distribution:

The **Uniform Distribution** describes a scenario where all outcomes within a specified range [a,b][*a*,*b*] are **equally likely**. It can be either:

- **Discrete** (e.g., rolling a fair die, where each integer outcome has the same probability).
- **Continuous** (e.g., selecting a random real number between a*a* and b*b*).

PDF for Uniform Distribution:

$$f(x) = \begin{cases} \frac{1}{b-a} & if\ x\ \in [a,b], \\ 0 & Otherwise \end{cases}$$

Where:

a = Lower bound (minimum value).

b = Upper bound (maximum value).

x = Continuous random variable in [a,b][a,b].

Exponential Distribution:

In a **Poisson process**, the **Exponential Distribution** is used to model the time that passes between individual events that are independent and happen at a fixed average rate..

Example: How far apart are customers arriving at a store?

PDF of the Exponential Distribution:

$$f(x) = \begin{cases} \lambda\Gamma e^{-\lambda x} & if\ x \geq 0, \\ 0, & if\ x < 0 \end{cases}$$

Where:

x = Time between events (a continuous random variable, x≥0x≥0).

λ (lambda) = Rate parameter (events per unit time).

CONDITIONAL PROBABILITY AND BAYES' THEOREM

Conditional Probability

Conditional probability refers to the probability of an event taking place if we already know that another event has happened.

Formula:

$$P\left(A/B\right) = \frac{P(A \cap B)}{P(B)} \quad (if\ P(B) > 0)$$

Where
($A|B$): Probability that iff event B happened event A might occur.
P(A∩B): Probability that A and B take place at the same time.
P(B)P: Probability that event B occurs.

Bayes' Theorem

Bayes' theorem relates the conditional and marginal probabilities of random events. It Updates the probability of an event based on new information.

Formula:

$$P\left(A/B\right) = \frac{P\left(B/A\right) \cdot P(A)}{P(B)}$$

Where:
P(A: Prior (initial belief about A).
$P(B|A)$: Likelihood (probability of observing B if A is true).
$P(A|B)$: Posterior (revised probability after observing B).
P(B): Marginal likelihood (total probability of B across all scenarios).

DESCRIPTIVE AND INFERENTIAL STATISTICS

Without descriptive and inferential statistics, data scientists would not be able to analyze their information or gather useful results.

DESCRIPTIVE STATISTICS

Storing information, as well as exploring it, is the main goal of descriptive statistics. They give a clear summary of the data that is helpful for exploring and analyzing information, as well as making decisions.

Measures of Central Tendency

Measures of central tendency explain what the average or typical value is in a set of data. They offer a simple summary for all the data in the dataset.

Table 11-1: Central tendency metrics for data analysis

Metric	Definition	Use in Data Science
Mean	The average value of a dataset.	- Summarizing data (e.g., average sales, average customer age).
		- Used as a baseline for predictive modeling and anomaly detection.
Median	The middle value in a sorted dataset.	- Robust to outliers, making it useful for skewed data (e.g., income distribution).
		- Used in scenarios where extreme values might distort the mean.
Mode	The most frequently occurring value.	- Identifying the most common category or value (e.g., most purchased product).
		- Useful for categorical data analysis.

Example 2: Computing Central Tendency

```
import numpy as np
data = [10, 20, 30, 40, 50, 50, 60]
print("Mean:", np.mean(data))
print("Median:", np.median(data))
from scipy.stats import mode
print("Mode:", mode(data).mode[0])
```

Measures of Dispersion

Measures of dispersion compare how far the individual observations are from the mean. They provide insights into the consistency and reliability of the data.

Table 11-2: Measures of dispersion

Metric	Definition	Use in Data Science
Variance	Measures how far each data point is from the mean.	- Understanding the spread of data (e.g., variability in customer spending).
		- Used in feature engineering and model evaluation.
Standard Deviation	The square root of variance, providing a measure of spread in the same units as the data.	- Assessing data consistency (e.g., consistency in delivery times).
		- Used in normalization and standardization of data.
Interquartile Range (IQR)	The range between the 25th percentile (Q1) and the 75th percentile (Q3).	- Identifying outliers (e.g., detecting anomalies in transaction amounts).
		- Robust to extreme values, making it useful for skewed datasets.

Data Visualization

We are already familiar with the term, data visualization. It is very important to convey our idea via pictures. Little wonder the saying goes, 'A picture is worth a thousand words'. There are various ways to visualize our dataset and results in statistics.

Table 11-3: Some visualization techniques and their use in datascience

Visualization Technique	Description	Use in Data Science
Histograms	Visualize the distribution of numerical data.	- Understanding data distribution (e.g., distribution of customer ages).
		- Identifying patterns and outliers.
Box Plots	Display the spread and skewness of data using quartiles.	- Comparing distributions across categories (e.g., sales by region).
		- Detecting outliers and understanding data variability.
Kernel Density Plots	Smooth representation of data distribution.	- Visualizing the probability density of continuous variables.
		- Useful for understanding the shape of data distributions.
Skewness and Kurtosis	Skewness: Measures the asymmetry of the	- Identifying non-normal distributions (e.g., skewed

Visualization Technique	Description	Use in Data Science
	data distribution.	income data).
	Kurtosis: Measures the "tailedness" of the distribution.	- Guiding data transformation and preprocessing steps.

Example 3: Computing Dispersion

```
print("Variance:", np.var(data))
print("Standard Deviation:", np.std(data))
print("Range:", np.ptp(data))
```

INFERENTIAL STATISTICS

Inferential statistics allow data scientists to make predictions or inferences about a population based on a sample of data. These techniques are critical for hypothesis testing, model evaluation, and decision-making.

Table 11-4: Sampling and Estimation

Concept	Definition	Use in Data Science
Population vs. Sample	**Population**: The entire set of data. **Sample**: A subset of the population.	- Analyzing samples from large datasets. - Generalizing findings to the population.
Central Limit Theorem (CLT)	States that the sampling distribution of the mean approaches a normal distribution as the sample size increases.	- Justifies the use of normal distribution in hypothesis testing. - Enables confidence interval estimation.

Table 11-5: Hypothesis Testing

Concept	Definition	Use in Data Science
Null and Alternative Hypotheses	**Null Hypothesis (H_0)**: Assumes no effect or difference. **Alternative Hypothesis (H_1)**: Assumes an effect or difference exists. **p-value**: Measures evidence against H_0; if $p < 0.05$, reject H_0.	- Testing the effectiveness of a new feature or model. - Validating assumptions about data.
Type I and Type II Errors	Type **I Error**: Rejecting H_0 when it is true (false positive). **Type II Error**: Failing to reject H_0 when it is false (false negative).	- Balancing the trade-off between false positives and false negatives (e.g., in fraud detection).
p-values and Significance Levels	**p-value**: Probability of observing the data given that H_0 is true. **Significance Level (α)**: The threshold for rejecting H_0 (commonly 0.05).	- Determining statistical significance of results (e.g., A/B testing).

Common Statistical Tests

1. t-tests: Compare the means of two groups.

Types:

- One-sample t-test: Compare a sample mean to a known value.

- Two-sample t-test: Compare means of two independent samples.

- Paired t-test: Compare means of the same group at different times.

Use in Data Science:

- Checking how much users interact with the new feature.

Example: Performing a t-test

```
from scipy.stats import ttest_ind

data1 = np.random.normal(50, 10, 100)
data2 = np.random.normal(55, 10, 100)
stat, p_value = ttest_ind(data1, data2)
print("p-value:", p_value)
```

2. Chi-square Tests: Assess the connection between different types of variables.

Types:

- Goodness-of-fit: Determine if the data we have in our experiment fits what is expected.

- Independence: Analyze if two categorical variables are independent of each other.

Use in Data Science:

- Reviewing survey results (e.g., surveying people's preferences by gender).

3. ANOVA (Analysis of Variance): Check the differences between outcomes for three or more groups.

Use in Data Science:

- Assessing how well several marketing techniques are performing.

APPLICATIONS OF DESCRIPTIVE AND INFERENTIAL STATISTICS IN DATA SCIENCE

1. Exploratory Data Analysis (EDA): Watch for patterns by using statistical measures and charts in Exploratory Data Analysis (EDA).
2. **Model Evaluation:** To confirm if the model is well-suited and carries importance, use inferential statistics.
3. **A/B Testing:** Test the two versions of a product or feature to determine which one performs better.
4. **Feature Engineering:** Find the most significant features for a machine learning model by using statistical analysis.
5. Decision-Making: Use statistics to base your decisions in business and research on solid data.

CASE STUDY: Using Descriptive and Inferential Statistics in Data Science

Problem Statement

A retail business is interested in understanding customer buying patterns to improve its strategies for promoting the company. They are looking for something in particular:
1. Summarize and depict the spending habits of your customers.
2. Test ideas about variations in spending by men and women.
3. Create a model that can help you recognize customers who spend a lot of money.

Dataset

These are the following columns in the dataset:
`CustomerID`: The CustomerID is a distinctive number to identify every customer.
`Gender`: Information on whether the customer is a man or a woman.

`Age`: Age of the customer.
`AnnualIncome`: Annual income is the money the customer receives in one year (measured in thousands of dollars).
`SpendingScore`: It measures how much a customer spends and assigns a number between 0 and 100.

Step 1: Descriptive Statistics and Data Visualization
We are going to use both descriptive statistics and visual aids to analyze the data.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

#Load the dataset
df = pd.read_csv("customer_data.csv")

#Display the first few rows
print(df.head())

#Descriptive statistics
print(df.describe())

#Distribution of SpendingScore
plt.figure(figsize=(8, 6))
sns.histplot(df['SpendingScore'], kde=True)
plt.title("Distribution of Spending Score")
plt.xlabel("Spending Score")
plt.ylabel("Frequency")
plt.show()
```

```
#Box plot of SpendingScore by Gender
plt.figure(figsize=(8, 6))
sns.boxplot(x='Gender', y='SpendingScore', data=df)
plt.title("Spending Score by Gender")
plt.xlabel("Gender")
plt.ylabel("Spending Score")
```

```
plt.show()`
```

Insights

1. The mean `SpendingScore` is 50.

2. The distribution of `SpendingScore` is approximately normal.

3. There is no significant difference in `SpendingScore` between male and female customers based on the box plot.

Step 2: Inferential Statistics - Hypothesis Testing

We will test whether there is a significant difference in `SpendingScore` between male and female customers using a two-sample t-test.

Hypotheses

- Null Hypothesis (H_0): There is no difference in mean `SpendingScore` between male and female customers.
- Alternative Hypothesis (H_1): There is a difference in mean `SpendingScore` between male and female customers.

```
from scipy.stats import ttest_ind

#Split data by gender
male_scores = df[df['Gender'] == 'Male']['SpendingScore']
female_scores = df[df['Gender'] == 'Female']['SpendingScore']

#Perform t-test
t_stat, p_value = ttest_ind(male_scores, female_scores)
print(f"T-statistic: {t_stat}, P-value: {p_value}")

#Interpret results
alpha = 0.05
if p_value < alpha:
   print("Reject the null hypothesis: There is a significant difference in spending scores between genders.")
else:
```

```
  print("Fail to reject the null hypothesis: There is no significant difference in spending scores between genders.")
```

Results

- T-statistic: 0.15, P-value: 0.88
- Since the p-value > 0.05, we fail to reject the null hypothesis.
- Conclusion: There is no significant difference in `SpendingScore` between male and female customers.

Step 3: Predictive Modeling

We will build a logistic regression model to predict whether a customer has a high `SpendingScore` (above 70).

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

#Create a binary target variable
df['HighSpending'] = df['SpendingScore'] > 70

#Select features and target
X = df[['Age', 'AnnualIncome']]
y = df['HighSpending']

#Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Train a logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

#Make predictions
y_pred = model.predict(X_test)

#Evaluate the model
print("Confusion Matrix:")
```

```
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Results

- The model achieves an accuracy of 85%.
- The confusion matrix and classification report show good performance in predicting high-spending customers.

Conclusion

1. Descriptive Statistics and Visualization:

- Helped summarize and visualize customer spending patterns.

- Identified no significant difference in spending between genders.

2. Inferential Statistics:

- Used a t-test to confirm no significant difference in spending between genders.

3. Predictive Modeling:

- Built a logistic regression model to predict high-spending customers with 85% accuracy.

This case study illustrates that using both descriptive and inferential statistics, along with machine learning, offers helpful insights to business leaders for making choices.

The complete code

```
Import libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import ttest_ind
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

 Load the dataset
df = pd.read_csv("customer_data.csv")

 Descriptive statistics and visualization
print(df.describe())
sns.histplot(df['SpendingScore'], kde=True)
plt.title("Distribution of Spending Score")
plt.show()

sns.boxplot(x='Gender', y='SpendingScore', data=df)
plt.title("Spending Score by Gender")
plt.show()

 Hypothesis testing
male_scores = df[df['Gender'] == 'Male']['SpendingScore']
female_scores = df[df['Gender'] == 'Female']['SpendingScore']
t_stat, p_value = ttest_ind(male_scores, female_scores)
print(f"T-statistic: {t_stat}, P-value: {p_value}")

 Predictive modeling
df['HighSpending'] = df['SpendingScore'] > 70
X = df[['Age', 'AnnualIncome']]
y = df['HighSpending']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

APPLICATION OF PROBABILITY IN DATA SCIENCE

1. Probability in Machine Learning

When making predictions, machines in machine learning need probability to account for uncertain information.

Bayesian Inference: Bayesian Inference refers to a technique that updates probabilities using Bayes' theorem.

Use in Data Science:

- Spam Detection: Check the likelihood that an email is spam by considering its accompanying keywords.
- Recommendation Systems: Update what each user shows an interest in from their behaviors.

Example:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

#Example: Spam detection
emails = ["win money now", "hello how are you", "free entry in contest"]
labels = ["spam", "not spam", "spam"]

#Convert text to features
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(emails)

#Train a Naive Bayes model
model = MultinomialNB()
model.fit(X, labels)

#Predict
new_email = ["free money"]
print(model.predict(vectorizer.transform(new_email)))   Output: ['spam']
```

Probabilistic Models: For example, techniques that allow for randomness and uncertainty in modeling, beans GMMs (GMMs) and HMMs (HMMs).

Use in Data Science:

- Anomaly Detection: Identify strange events happening within the data.
- Natural Language Processing: Focus on shaping sequences of words or single characters.

2. Statistical Analysis in Data Science

For understanding information, testing ideas and deciding based on data, statistical analysis is needed.

Exploratory Data Analysis (EDA): A method to study data by summarizing and plotting to identify patterns, trends and exceptions.

Use in Data Science:

- Data Cleaning: Find information that is missing from the dataset.
- Feature Engineering: Use statistics to develop additional features.

Example:

```
import seaborn as sns
import matplotlib.pyplot as plt

 Load dataset
df = sns.load_dataset("titanic")

 Visualize age distribution
sns.histplot(df['age'].dropna(), kde=True)
plt.title("Age Distribution of Titanic Passengers")
plt.show()
```

A/B Testing and Experimentation: A statistical way to compare two versions to see which one is more successful is called A/B Testing and Experimentation.

Use in Data Science:

- Marketing: Compare the outcomes of various ads..

- Product Development: Check how the addition of new features influences user activity.

Example:

```
from scipy.stats import ttest_ind
 Example: A/B test for website conversion rates
group_a = [0, 1, 0, 1, 1, 0, 0, 1, 0, 1]  Control group
group_b = [1, 1, 0, 1, 1, 1, 0, 1, 1, 1]  Treatment group

 Perform t-test
t_stat, p_value = ttest_ind(group_a, group_b)
print(f"P-value: {p_value}")  Output: P-value: 0.049 (significant at alpha=0.05)
```

3. Real-World Use Cases

Risk Assessment : Now, chances of negative outcomes can be measured using probabilities.

Use in Data Science:

- Finance: Evaluate the chances of funds not being repaid or lost.
- Healthcare: Try to identify the risk of diseases or complications happening.

Example:

```
# Example: Predicting loan default risk
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

 Simulated data
X = [[25, 50000], [30, 60000], [35, 70000], [40, 80000]]  Age, Income
y = [0, 0, 1, 1]  0: No default, 1: Default

 Train a model
model = RandomForestClassifier()
model.fit(X, y)

 Predict
```

```
new_customer = [[28, 55000]]
print(model.predict(new_customer))   Output: [0] (no default)
```

Predictive Modeling: By using various methods, it is possible to make predictions for the coming days or weeks.

Use in Data Science:

- Sales Forecasting: Predict future sales based on historical data.
- Customer Churn: Identify customers likely to stop using a service.

Example:

```
from sklearn.linear_model import LinearRegression
import numpy as np

 Example: Sales forecasting
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)   Months
y = np.array([100, 200, 300, 400, 500])        Sales

 Train a linear regression model
model = LinearRegression()
model.fit(X, y)

 Predict
future_month = np.array([6]).reshape(-1, 1)
print(model.predict(future_month))   Output: [600]
```

Decision-Making: Using statistical insights to make informed decisions.

Use in Data Science:

- Business Strategy: Optimize pricing, marketing, and operations.
- Public Policy: Evaluate the impact of policy changes.

Example:

```
Example: Optimizing pricing strategy
 import numpy as np
```

```
 Simulated data
prices = [10, 20, 30, 40, 50]
demand = [100, 80, 60, 40, 20]
revenue = np.array(prices) * np.array(demand)

 Find optimal price
optimal_price = prices[np.argmax(revenue)]
print(f"Optimal Price: ${optimal_price}")   Output: Optimal Price: $20
```

QUESTIONS

1. Define probability and explain its importance in data science.
2. Differentiate between **discrete** and **continuous** probability distributions.
3. Explain the concepts of **Type I Error** and **Type II Error** in hypothesis testing.
4. What is the significance of **confidence intervals** in inferential statistics?
5. Discuss the differences between **descriptive** and **inferential** statistics.
6. Describe a real-world application where hypothesis testing is useful in data science.
7. Generate a dataset of 1000 random numbers from a normal distribution with a mean of 50 and a standard deviation of 10 using NumPy. Compute and print the **mean, median, variance, and standard deviation** of the dataset.
8. Load a sample dataset (e.g., iris or titanic from seaborn) and compute the **correlation matrix** between numerical features.
9. Create a histogram and boxplot for a dataset of your choice using matplotlib or seaborn. Interpret the distribution of the data.
10. Using Python, simulate the rolling of two six-sided dice 10,000 times and plot the distribution of their sum.
11. Generate a **binomial distribution** where the probability of success is 0.3 in 10 trials. Plot its probability mass function (PMF).

12. Write a Python function to compute the **Bayes' Theorem** given prior probability, likelihood, and marginal probability.
13. Perform a **one-sample t-test** on a dataset to check if the sample mean significantly differs from a given population mean.
14. Conduct a **two-sample t-test** to compare the means of two independent groups from a dataset.
15. Perform a **Chi-Square test** to determine if the two categorical variables in a dataset (such as titanic: survival and gender) are not related.
16. Analyze a data set with three or more groups using an **ANOVA test** and draw conclusions.
17. Create a dataset and use the scipy.stats module to find a **95% confidence interval** for the mean. .
18. Execute a **linear regression analysis** for a dataset and check the importance of the regression coefficients by testing different hypotheses.

MODULE 12

ADVANCED PANDAS FOR DATA SCIENCE

Dealing with data in Python often relies on Pandas. While the essential features are enough for many purposes, to efficiently work with big and complicated data, you need to know the advanced tools. This module is based on what you have learned about Pandas and teaches you more advanced methods for data analysis.

As a result of this module, you will be able to:

- Efficiently manipulate large datasets using Pandas.
- Utilize advanced indexing techniques.
- Apply group operations and time series analysis.
- Optimize Pandas performance for large-scale data processing.

By mastering the advanced techniques covered in this module, you will be well-prepared to handle complex data science tasks and optimize your data processing workflows using Pandas.

INTRODUCTION

This module discusses Pandas features such as multi-indexing, how to boost performance and handling of big data. At this point, we should have learned how to import pandas.

```
In [1]: import pandas as pd
```

As a result, when you see a pd. in code, it's referring to pandas. You may also find it easier to import Series and DataFrame into the local namespace since they are so frequently used:

```
In [2]: from pandas import Series, DataFrame
```

To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While

they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its *index*.
The simplest Series is formed from only an array of data:

```
In [11]: obj = pd.Series([4, 7, -5, 3])
In [12]: obj
```

```
Out[12]:
0 4
1 7
2 -5
3 3
dtype: int64
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:

```
In [13]: obj.values
Out[13]: array([ 4, 7, -5, 3])
In [14]: obj.index # like range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an index identifying each data point with a label:

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
In [16]: obj2
Out[16]:
d 4
b 7
```

```
a -5
c 3
dtype: int64
In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [18]: obj2['a']
Out[18]: -5
In [19]: obj2['d'] = 6
In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c 3
a -5
d 6
dtype: int64
```

Here ['c', 'a', 'd'] is interpreted as a list of indices, even though it contains strings instead of integers. Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [21]: obj2[obj2 > 0]
Out[21]:
d 6
b 7
c 3
dtype: int64
In [22]: obj2 * 2
Out[22]:
d 12
b 14
a -10
c 6
dtype: int64
In [23]: np.exp(obj2)
```

```
Out[23]:
d 403.428793
b 1096.633158
a 0.006738
c 20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

```
In [24]: 'b' in obj2
Out[24]: True
In [25]: 'e' in obj2
Out[25]: False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
In [27]: obj3 = pd.Series(sdata)
In [28]: obj3
Out[28]:
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
dtype: int64
```

When you are only passing a dict, the index in the resulting Series will have the dict's keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']
In [30]: obj4 = pd.Series(sdata, index=states)
In [31]: obj4
Out[31]:
California NaN
Ohio 35000.0
Oregon 16000.0
```

```
Texas 71000.0
dtype: float64
```

Here, three values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number), which is considered in pandas to mark missing or *NA* values. Since 'Utah' was not included in states, it is excluded from the resulting object.

I will use the terms “missing” or “NA” interchangeably to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data:

```
In [32]: pd.isnull(obj4)
Out[32]:
California True
Ohio False
Oregon False
Texas False
dtype: bool
In [33]: pd.notnull(obj4)
Out[33]:
California False
Ohio True
Oregon True
Texas True
dtype: bool
```

Series also has these as instance methods:

```
In [34]: obj4.isnull()
Out[34]:
California True
Ohio False
Oregon False
Texas False
dtype: bool
```

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:

```
In [35]: obj3
Out[35]:
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
dtype: int64
In [36]: obj4
Out[36]:
California NaN
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
dtype: float64
In [37]: obj3 + obj4
Out[37]:
California NaN
Ohio 70000.0
Oregon 32000.0
Texas 142000.0
Utah NaN
dtype: float64
```

Data alignment features will be addressed in more detail later. If you have experience with databases, you can think about this as being similar to a join operation.

Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

```
In [38]: obj4.name = 'population'
In [39]: obj4.index.name = 'state'
In [40]: obj4
Out[40]:
state
California NaN
```

```
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
Name: population, dtype: float64
```

A Series's index can be altered in-place by assignment:

```
In [41]: obj
Out[41]:
0 4
1 7
2 -5
3 3
dtype: int64
In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
In [43]: obj
Out[43]:
Bob 4
Steve 7
Jeff -5
Ryan 3
dtype: int64
```

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are outside the scope of this book. While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hierarchical indexing.

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [45]: frame
```

Out[45]:

pop	state	year
0	1.5	Ohio 2000
1	1.7	Ohio 2001
2	3.6	Ohio 2002
3	2.4	Nevada 2001
4	2.9	Nevada 2002
5	3.2	Nevada 2003

If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table. For large DataFrames, the head method selects only the first five rows:

```
In [46]: frame.head()
```

Out[46]:

pop	state	year
0	1.5	Ohio 2000
1	1.7	Ohio 2001
2	3.6	Ohio 2002
3	2.4	Nevada 2001
4	2.9	Nevada 2002

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

```
Out[47]:
year state pop
0 2000 Ohio 1.5
1 2001 Ohio 1.7
2 2002 Ohio 3.6
3 2001 Nevada 2.4
4 2002 Nevada 2.9
5 2003 Nevada 3.2
```

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
....: index=['one', 'two', 'three', 'four',
....: 'five', 'six'])
In [49]: frame2
```

```
Out[49]:
year state pop debt
one 2000 Ohio 1.5 NaN
two 2001 Ohio 1.7 NaN
three 2002 Ohio 3.6 NaN
four 2001 Nevada 2.4 NaN
five 2002 Nevada 2.9 NaN
six 2003 Nevada 3.2 NaN
```

```
In [50]: frame2.columns
```

```
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [51]: frame2['state']
```

```
Out[51]:
one Ohio
two Ohio
three Ohio
four Nevada
five Nevada
six Nevada
Name: state, dtype: object
```

```
In [52]: frame2.year
Out[52]:
one 2000
two 2001
three 2002
four 2001
five 2002
six 2003
Name: year, dtype: int64
```

Attribute-like access (e.g., frame2.year) and tab completion of column names in IPython is provided as a convenience. frame2[column] works for any column name, but frame2.column only works when the column name is a valid Python variable name.

Note that the returned Series have the same index as the DataFrame, and their name attribute has been appropriately set. Rows can also be retrieved by position or name with the special loc attribute (much more on this later):

```
In [53]: frame2.loc['three']
Out[53]:
year 2002
state Ohio
pop 3.6
debt NaN
Name: three, dtype: object
```

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

```
In [54]: frame2['debt'] = 16.5
In [55]: frame2
Out[55]:
year state pop debt
one 2000 Ohio 1.5 16.5
two 2001 Ohio 1.7 16.5
```

```
three 2002 Ohio 3.6 16.5
four 2001 Nevada 2.4 16.5
five 2002 Nevada 2.9 16.5
six 2003 Nevada 3.2 16.5
```

```
In [56]: frame2['debt'] = np.arange(6.)
In [57]: frame2
```

```
Out[57]:
year state pop debt
one 2000 Ohio 1.5 0.0
two 2001 Ohio 1.7 1.0
three 2002 Ohio 3.6 2.0
four 2001 Nevada 2.4 3.0
five 2002 Nevada 2.9 4.0
six 2003 Nevada 3.2 5.0
```

When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
In [59]: frame2['debt'] = val
In [60]: frame2
```

```
Out[60]:
year state pop debt
one 2000 Ohio 1.5 NaN
two 2001 Ohio 1.7 -1.2
three 2002 Ohio 3.6 NaN
four 2001 Nevada 2.4 -1.5
five 2002 Nevada 2.9 -1.7
six 2003 Nevada 3.2 NaN
```

Assigning a column that doesn't exist will create a new column. The del keyword will delete columns as with a dict. As an example of del, I first add a new column of boolean values where the state column equals 'Ohio':

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'
In [62]: frame2
```

```
Out[62]:
```

year state pop debt eastern
one 2000 Ohio 1.5 NaN True
two 2001 Ohio 1.7 -1.2 True
three 2002 Ohio 3.6 NaN True
four 2001 Nevada 2.4 -1.5 False
five 2002 Nevada 2.9 -1.7 False
six 2003 Nevada 3.2 NaN False
New columns cannot be created with the frame2.eastern syntax.
The del method can then be used to remove this column:

```
In [63]: del frame2['eastern']
In [64]: frame2.columns
```

Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
The column returned from indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's copy method.
Another common form of data is a nested dict of dicts:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [66]: frame3 = pd.DataFrame(pop)
In [67]: frame3
```

Out[67]:
Nevada Ohio
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

```
In [68]: frame3.T
```

Out[68]:
2000 2001 2002

Nevada NaN 2.4 2.9
Ohio 1.5 1.7 3.6

The keys in the inner dicts are combined and sorted to form the index in the result.

This isn't true if an explicit index is specified:

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
```

Out[69]:
Nevada Ohio
2001 2.4 1.7
2002 2.9 3.6
2003 NaN NaN

Dicts of Series are treated in much the same way:

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
....: 'Nevada': frame3['Nevada'][:2]}
In [71]: pd.DataFrame(pdata)
```

Out[71]:
Nevada Ohio
2000 NaN 1.5
2001 2.4 1.7

If a DataFrame's index and columns have their name attributes set, these will also be displayed:

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'
In [73]: frame3
```

Out[73]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6

As with Series, the values attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [74]: frame3.values
```

Out[74]:
array([[nan, 1.5],
[2.4, 1.7],
[2.9, 3.6]])

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns:

```
In [75]: frame2.values
```

```
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
[2001, 'Ohio', 1.7, -1.2],
[2002, 'Ohio', 3.6, nan],
[2001, 'Nevada', 2.4, -1.5],
[2002, 'Nevada', 2.9, -1.7],
[2003, 'Nevada', 3.2, nan]], dtype=object)
```

Table 12-1: Possible Data Inputs to DataFrame Constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels.
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case.
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case.
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column

Type	Notes
	labels.
List of lists or tuples	Treated as the "2D ndarray" case.
Another DataFrame	The DataFrame's indexes are used unless different ones are passed.
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result.

Index Objects

pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])
In [77]: index = obj.index
In [78]: index
```

```
Out[78]: Index(['a', 'b', 'c'], dtype='object')
```

```
In [79]: index[1:]
```

```
Out[79]: Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can't be modified by the user:

```
index[1] = 'd' # TypeError
```

Immutability makes it safer to share Index objects among data structures:

```
In [80]: labels = pd.Index(np.arange(3))
In [81]: labels
```

```
Out[81]: Int64Index([0, 1, 2], dtype='int64')
```

```
In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)
In [83]: obj2
```

```
Out[83]:
```

```
0 1.5
1 -2.5
2 0.0
dtype: float64
```

```
In [84]: obj2.index is labels
```

```
Out[84]: True
```

Some users will not often take advantage of the capabilities provided by indexes, but because some operations will yield results containing indexed data, it's important to understand how they work. In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [85]: frame3
```

```
Out[85]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
```

```
In [86]: frame3.columns
```

```
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

```
In [87]: 'Ohio' in frame3.columns
```

```
Out[87]: True
```

```
In [88]: 2003 in frame3.index
```

```
Out[88]: False
```

Unlike Python sets, a pandas Index can contain duplicate labels:

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
In [90]: dup_labels
```

```
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label. Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in Table 12-2.

Table 12-2: Some Index Methods and Properties

Method	Description	Example	Output
append()	Concatenates with additional Index objects, producing a new Index.	idx1.append(idx2)	Index([1, 2, 3, 4, 5, 3, 4, 5, 6, 7])
difference()	Computes the set difference between two Index objects.	idx1.difference(idx2)	Index([1, 2])
intersection()	Computes the set intersection between two Index objects.	idx1.intersection(idx2)	Index([3, 4, 5])
union()	Computes the set union of two Index objects.	idx1.union(idx2)	Index([1, 2, 3, 4, 5, 6, 7])
isin()	Returns a boolean array indicating whether each value is in the passed collection.	idx1.isin([3, 5, 7])	[False, False, True, False, True]
delete()	Computes a new Index with the element at index i removed.	idx1.delete(2)	Index([1, 2, 4, 5])
drop()	Computes a new Index by deleting specified values.	idx1.drop([3, 5])	Index([1, 2, 4])
insert()	Inserts an element at a specified index, creating a new Index.	idx1.insert(1, 10)	Index([1, 10, 2, 3, 4, 5])

Method	Description	Example	Output
is_monotonic	Returns True if each element is greater than or equal to the previous element.	idx1.is_monotonic	True
is_unique	Returns True if the Index has no duplicate values.	idx1.is_unique	True
unique()	Returns an array of unique values in the Index.	idx1.unique()	array([1, 2, 3, 4, 5])

SELECTING DATA WITH ILOC AND LOC:

UNDERSTANDING iLOC AND LOC

In **pandas**, .iloc and .loc are used for selecting and filtering data from a **DataFrame** or **Series**. They provide powerful ways to access specific rows and columns based on different indexing methods.

1. iloc (Integer-location based indexing)

.iloc, the no-nonsense, straight-to-the-point gatekeeper of pandas! If .loc is like that friendly shopkeeper who lets you use names to find your garri or maggi, .iloc is the strict teacher who insists you call your classmates by *number*, not name. No long talk

Imagine you're in a Danfo (Lagos-style), and the conductor is shouting, "First row, second seat!" , that's .iloc[0, 1] for you. It doesn't care if your name is Chinedu or Ada, all it wants to know is **which number** you are sitting on. The first row is 0, the second row is 1, and so on

So when you say df.iloc[2], you're telling pandas, "Please, give me the **third** row of this table" (because counting starts from 0). Just

like when you want to get served jollof rice at a party, .iloc doesn't ask for titles or descriptions , it just dives in based on *position*. Clean, sharp.

The syntax follows:

df.iloc[row_index, column_index]

- Supports slicing and list-based selection.

Examples:

```
import pandas as pd
# Sample DataFrame
data = {'A': [10, 20, 30], 'B': [40, 50, 60], 'C': [70, 80, 90]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

# Selecting a single value
print(df.iloc[1, 2])  # Output: 80 (row2, column C)

# Selecting an entire row
print(df.iloc[1])  # Output: Second row

# Selecting multiple rows and columns
print(df.iloc[0:2, 1:3])  # Rows 0 to 1, Columns 1 to 2
```

2. loc (Label-based indexing)

Now, .loc is the friendly, cool cousin of .iloc who loves to make life easier with names. If .iloc is all about the numbers, .loc is all about labels. It's like you're walking into your auntie's party, and she says, "Chinedu, you're sitting at the table near the window!" No need to ask the seat number, just use your name!

Imagine, instead of saying "Give me row 2, column 1," with .loc, you can say, "Give me the data for Chinedu and Ada" if those are the row labels. It's all about calling things by their rightful names! And the best part? You're not forced to use zero-based numbering, which can be a headache

For example, if you have a DataFrame and want to access the row with the label "Chinedu", you'd do df.loc['Chinedu']. No need to count or guess, just use the label! It's like calling your friend to meet you at the local suya joint: .loc is especially useful when you have a well-defined set of labels for your rows or columns. It's like that VIP entrance where your name is on the list, no questions asked. Whether you're dealing with dates, IDs, or names, just call them out directly.

The syntax follows:

df.loc[row_label, column_label]

- Supports slicing and boolean conditions.

Examples:

```
# Selecting a single value
print(df.loc['row2', 'C'])  # Output: 80

# Selecting a row
print(df.loc['row3'])  # Output: Third row

# Selecting multiple rows and columns
print(df.loc[['row1', 'row3'], ['A', 'C']])
```

Table 12-3: iloc vs. loc: **Positional vs. Label-Based Selection in Pandas**

Feature	**iloc (Integer-based)**	**loc (Label-based)**
Index Type	Uses integer positions (0,1,2...)	Uses explicit labels ('row1', 'row2'...)
Slicing	Exclusive end (df.iloc[0:2] selects rows 0 & 1)	Inclusive end (df.loc['row1':'row2'] selects both rows)
Supports Lists	Yes (df.iloc[[0, 2]])	Yes (df.loc[['row1', 'row3']])

Feature	iloc (Integer-based)	loc (Label-based)
Boolean Masking	Not directly	Yes (df.loc[df['A'] > 20])

SUMMARIZING AND COMPUTING DESCRIPTIVE STATISTICS

Pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame:

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
.....: [np.nan, np.nan], [0.75, -1.3]],
.....: index=['a', 'b', 'c', 'd'],
.....: columns=['one', 'two'])
In [231]: df
```

```
Out[231]:
one two
a 1.40 NaN
b 7.10 -4.5
c NaN NaN
d 0.75 -1.3
```

Calling DataFrame's sum method returns a Series containing column sums:

```
In [232]: df.sum()
```

```
Out[232]:
one 9.25
two -5.80
dtype: float64
```

Passing axis='columns' or axis=1 sums across the columns instead:

```
In [233]: df.sum(axis='columns')
```

```
Out[233]:
a 1.40
b 2.60
c NaN
d -0.55
dtype: float64
```

NA values are excluded unless the entire slice (row or column in this case) is NA.

This can be disabled with the skipna option:

```
In [234]: df.mean(axis='columns', skipna=False)
```

```
Out[234]:
a NaN
b 1.300
c NaN
d -0.275
dtype: float64
```

Table 12-4: Options for Reduction Methods

Method	Description
Axis	Axis to reduce over; `0` for DataFrame's rows and `1` for columns.
Skipna	Exclude missing values; `True` by default.
Level	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex).

Some methods, like idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [235]: df.idxmax()
```

```
Out[235]:
one b
two d
dtype: object
```

Other methods are *accumulations*:

```
In [236]: df.cumsum()
```

```
Out[236]:
one two
a 1.40 NaN
b 8.50 -4.5
c NaN NaN
d 9.25 -5.8
```

Another type of method is neither a reduction nor an accumulation. describe is one such example, producing multiple summary statistics in one shot:

```
In [237]: df.describe()
```

```
Out[237]:
one two
count 3.000000 2.000000
mean 3.083333 -2.900000
std 3.493685 2.262742
min 0.750000 -4.500000
25% 1.075000 -3.700000
50% 1.400000 -2.900000
75% 4.250000 -2.100000
max 7.100000 -1.300000
```

On non-numeric data, describe produces alternative summary statistics:

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
In [239]: obj.describe()
```

```
Out[239]:
count 16
unique 3
top a
freq 8
dtype: object
```

Table 12-5: **Descriptive and Summary Statistics with Examples and Output**

Method	Description	Example	Expected Output
Count	Number of non-NA values.	df.count()	A: 5, B: 5
Describe	Summary statistics for Series or DataFrame columns.	df.describe()	Count, Mean, Std, Min, 25%, 50%, 75%, Max for each column (see below).
min, max	Compute min and max values.	df['A'].min(), df['A'].max()	10, 50
argmin, argmax	Index locations (integers) of min/max values.	df['A'].argmin(), df['A'].argmax()	0, 4
idxmin, idxmax	Index labels of min/max values.	df['A'].idxmin(), df['A'].idxmax()	0, 4
Quantile	Compute quantile (0-1 range).	df['A'].quantile(0.75)	40.0
Sum	Sum of values.	df['A'].sum()	150
Mean	Mean of values.	df['B'].mean()	25.0

Method	Description	Example	Expected Output
Median	Median (50% quantile) of values.	df['A'].median()	30.0
Mad	Mean absolute deviation from mean.	df['A'].mad()	12.0
Prod	Product of all values.	df['A'].prod()	12000000
Var	Sample variance.	df['A'].var()	250.0
Std	Sample standard deviation.	df['A'].std()	15.81
Skew	Sample skewness (third moment).	df['A'].skew()	0.0
Kurt	Sample kurtosis (fourth moment).	df['A'].kurt()	-1.2
Cumsum	Cumulative sum.	df['A'].cumsum()	[10, 30, 60, 100, 150]
cummin, cummax	Cumulative min/max.	df['A'].cummin(), df['A'].cummax()	[10, 10, 10, 10, 10], [10, 20, 30, 40, 50]
Cumprod	Cumulative	df['A'].cumprod()	[10, 200, 6000,

Method	Description	Example	Expected Output
	product.		240000, 12000000]
Diff	First arithmetic difference.	df['A'].diff()	[NaN, 10, 10, 10, 10]
pct_change	Compute percent changes.	df['A'].pct_change()	[NaN, 1.0, 0.5, 0.333, 0.25]

Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance using the add-on pandas-datareader package. If you don't have it installed already, it can be obtained via conda or pip:

```
conda install pandas-datareader
```

I use the pandas_datareader module to download some data for a few stock tickers:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
for ticker, data in all_data.items()})
```

It's possible by the time you are reading this that Yahoo! Finance no longer exists since Yahoo! was acquired by Verizon in 2017. Refer to the pandas-datareader documentation online for the latest functionality.

I now compute percent changes of the prices, a time series operation which will be explored further in this book.

```
In [242]: returns = price.pct_change()
In [243]: returns.tail()
```

```
Out[243]:
AAPL GOOG IBM MSFT
Date
2016-10-17 -0.000680 0.001837 0.002072 -0.003483
2016-10-18 -0.000681 0.019616 -0.026168 0.007690
2016-10-19 -0.002979 0.007846 0.003583 -0.002255
2016-10-20 -0.000512 -0.005652 0.001719 -0.004867
2016-10-21 -0.003930 0.003011 -0.012474 0.042096
```

The corr method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, cov computes the covariance:

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144
In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

Since MSFT is a valid Python attribute, we can also select these columns using more concise syntax:

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

DataFrame's corr and cov methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [247]: returns.corr()
```

```
Out[247]:
AAPL GOOG IBM MSFT
AAPL 1.000000 0.407919 0.386817 0.389695
GOOG 0.407919 1.000000 0.405099 0.465919
IBM 0.386817 0.405099 1.000000 0.499764
MSFT 0.389695 0.465919 0.499764 1.000000
```

```
In [248]: returns.cov()
```

```
Out[248]:
AAPL GOOG IBM MSFT
```

```
AAPL 0.000277 0.000107 0.000078 0.000095
GOOG 0.000107 0.000251 0.000078 0.000108
IBM 0.000078 0.000078 0.000146 0.000089
MSFT 0.000095 0.000108 0.000089 0.000215
```

Using DataFrame's corrwith method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [249]: returns.corrwith(returns.IBM)
```

```
Out[249]:
AAPL 0.386817
GOOG 0.405099
IBM 1.000000
MSFT 0.499764
dtype: float64
```

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

```
In [250]: returns.corrwith(volume)
```

```
Out[250]:
AAPL -0.075565
GOOG -0.007067
IBM -0.204849
MSFT -0.092950
dtype: float64
```

Passing axis='columns' does things row-by-row instead. In all cases, the data points are aligned by label before the correlation is computed.

Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is unique, which gives you an array of the unique values in a Series:

```
In [252]: uniques = obj.unique()
In [253]: uniques
```

```
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (uniques.sort()). Relatedly, value_counts computes a Series containing value frequencies:

```
In [254]: obj.value_counts()
```

```
Out[254]:
c 3
a 3
b 2
d 1
dtype: int64
```

The Series is sorted by value in descending order as a convenience. value_counts is also available as a top-level pandas method that can be used with any array or sequence:

```
In [255]: pd.value_counts(obj.values, sort=False)
```

```
Out[255]:
a 3
b 2
c 3
d 1
dtype: int64
```

isin performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

```
In [256]: obj
```

```
Out[256]:
0 c
1 a
2 d
3 a
4 a
```

```
5 b
6 b
7 c
8 c
dtype: object
```

```
In [257]: mask = obj.isin(['b', 'c'])
In [258]: mask
```

```
Out[258]:
0 True
1 False
2 False
3 False
4 False
5 True
6 True
7 True
8 True
dtype: bool
```

```
In [259]: obj[mask]
```

```
Out[259]:
0 c
5 b
6 b
7 c
8 c
dtype: object
```

Related to isin is the Index.get_indexer method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
In [261]: unique_vals = pd.Series(['c', 'b', 'a'])
In [262]: pd.Index(unique_vals).get_indexer(to_match)
```

```
Out[262]: array([0, 2, 1, 1, 0, 2])
```

See Table 12-6 for a reference on these methods.

Table 12-6: Unique, Value Counts, and Set Membership Methods

Method	Description	Example	Expected Output
Isin	Compute boolean array indicating whether each Series value is contained in the passed sequence.	df['A'].isin([10, 30, 50])	[True, False, True, False, True]
Match	Compute integer indices for each value in an array into another array of distinct values (useful for data alignment and joins).	df['A'].map({10: 0, 20: 1, 30: 2, 40: 3, 50: 4})	[0, 1, 2, 3, 4]
Unique	Compute array of unique values in a Series, returned in the order observed.	df['A'].unique()	[10, 20, 30, 40, 50]
value_counts	Return a Series containing unique values as its index and frequencies as its values, sorted in descending order.	df['A'].value_counts()	{10:1, 20:1, 30:1, 40:1, 50:1}

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....: 'Qu2': [2, 3, 1, 2, 3],
.....: 'Qu3': [1, 5, 2, 4, 4]})
In [264]: data
```

```
Out[264]:
Qu1 Qu2 Qu3
0 1 2 1
1 3 3 5
2 4 1 2
3 3 2 4
4 4 3 4
```

Passing pandas.value_counts to this DataFrame's apply function gives:

```
In [265]: result = data.apply(pd.value_counts).fillna(0)
In [266]: result
```

```
Out[266]:
Qu1 Qu2 Qu3
1 1.0 1.0 1.0
2 0.0 2.0 1.0
3 2.0 2.0 0.0
4 2.0 0.0 2.0
5 0.0 0.0 1.0
```

Here, the row labels in the result are the distinct values occurring in all of the columns. The values are the respective counts of these values in each column.

ADVANCED INDEXING TECHNIQUES

MultiIndex

Multi-indexing allows hierarchical indexing of DataFrames, useful for handling multi-dimensional data. It allows you to work with higher-dimensional data in a 2D DataFrame:

Creating MultiIndex:

Use pd.MultiIndex.from_arrays() or pd.MultiIndex.from_tuples() t o create hierarchical indices.

```
# Create MultiIndex
```

```
arrays = [['A', 'A', 'B', 'B'], [1, 2, 1, 2]]
index = pd.MultiIndex.from_arrays(arrays, names=('group', 'number'))
df = pd.DataFrame({'value': [10, 20, 30, 40]}, index=index)
print(df)
```

Example

```
import pandas as pd
import numpy as np
arrays = [
  ['A', 'A', 'B', 'B'],
  [1, 2, 1, 2]
]
index = pd.MultiIndex.from_arrays(arrays, names=('letter', 'number'))
df = pd.DataFrame(np.random.randn(4, 2), index=index, columns=['Value1', 'Value2'])
print(df)
```

Indexing and Slicing:

Access data using .loc[] and .iloc[] with MultiIndex. Use pd.IndexSlice for more complex slicing operations.

```
# Access data using MultiIndex
print(df.loc['A', 1])
# Slicing with pd.IndexSlice
idx = pd.IndexSlice
print(df.loc[idx['A', :]])
```

Cross-sections:

Extract cross-sections of data using pd.DataFrame.xs().

```
# Cross-section
print(df.xs(key=1, level='number'))
```

EFFICIENT DATA MANIPULATION WITH PANDAS HANDLING LARGE DATASETS

Working with large datasets can be challenging due to memory constraints and processing time. Pandas provides several techniques to handle such datasets efficiently:

Chunking:

When reading large files, you can process the data in smaller chunks using the chunksize parameter in pd.read_csv(). This allows you to work with data that doesn't fit into memory all at once.

```
import pandas as pd

# Read a large CSV file in chunks
chunk_size = 100000
chunks = pd.read_csv('large_dataset.csv', chunksize=chunk_size)

# Process each chunk
for chunk in chunks:
    # Perform operations on each chunk
    print(chunk.head())
```

Example

```
chunk_size = 1000
for chunk in pd.read_csv('large_dataset.csv', chunksize=chunk_size):
    process(chunk)  # Replace with actual processing logic
```

Dask Integration:

Dask helps Pandas to manage large datasets that do not fit in memory. The dask.dataframe sees to it that data analysis resembles methods in Pandas, even though it takes place in parallel.

```
import dask.dataframe as dd
# Read a large CSV file using Dask
df = dd.read_csv('large_dataset.csv')

# Perform operations
df['column'] = df['column'] * 2
result = df.compute()  # Converts Dask DataFrame to Pandas DataFrame
print(result.head())
```

Memory Optimization:

Improve the efficiency of your program by changing columns with floating-point data to the shorter (i.e., float32, instead of float64) format and converting columns with repeated values to categorical data..

```
# Convert data types to save memory
df['column'] = df['column'].astype('float32')

# Use categorical data for repetitive values
df['category_column'] = df['category_column'].astype('category')
```

Example

```
df['C'] = df['C'].astype('float32')
print(df.info())
```

ADVANCED DATA CLEANING

When working with data, cleaning it is very important. Advanced ways include:

Handling Missing Data:

You may use functions such as interpolate() or KNN imputation to deal with missing values.

```
# Interpolate missing values
df['column'].interpolate(method='linear', inplace=True)

# KNN Imputation
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=2)
df_filled = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

Data Transformation:

Convert your data by calling pd.melt() to change it to long form or use pd.pivot_table() to compact and modify your data.

```
# Melt DataFrame
melted_df = pd.melt(df, id_vars=['id'], value_vars=['col1', 'col2'])
```

```
# Pivot Table
pivot_df = pd.pivot_table(df, values='value', index='date', columns='category
')
```

String Operations:

When you want to extract substrings, replace patterns or split your columns in pandas, use the pd.Series.str accessor.

```
# Extract substrings
df['new_column'] = df['string_column'].str.extract(r'(\d+)')

# Replace patterns
df['string_column'] = df['string_column'].str.replace('old', 'new')

# Split columns
df[['first', 'last']] = df['name'].str.split(' ', expand=True)
```

BOOLEAN INDEXING AND QUERY METHOD

Boolean Indexing:

It is simple to use **Boolean Indexing** in pandas to get precise results when working with data. It means you use a set of guidelines to select rows from a DataFrame that fit what you have set. Basically, by setting a condition, you can get rid of the rows that do not meet the needed criteria.

```
# Boolean indexing
filtered_df = df[df['value'] > 20]
print(filtered_df)
```

Query Method:

With the **query()** method, you can filter data more conveniently and quickly than by using regular boolean indexing. You can use a string expression to apply conditions in your DataFrame which can make your code appear simpler and more manageable when you deal with many conditions.

```
# Query method
filtered_df = df.query('value > 20')
print(filtered_df)
```

GROUP OPERATIONS AND AGGREGATIONS

GroupBy Operations

Using GroupBy, you can group records, execute functions on them and combine the final data into a new list.

GroupBy Mechanics:

Understand how data is split, processed by a function and combined using groupby().

```
# GroupBy example
grouped = df.groupby('group')
print(grouped.mean())
```

Aggregation Functions:

After using the groupby() method to organize data, **aggregation functions** in pandas allow you to summarize the results and examine them. They allow you to use statistical methods such as sum, mean, count and so on with every group, making useful observations from big databases..

Common Built-in Aggregation Functions:

- **sum()**: This function adds up all the values within each group.
- **mean()**: It calculates the average of values within each group.
- **count()**: This counts the number of non-null values in each group.
- **min()**: Finds the minimum value in each group.
- **max()**: Finds the maximum value in each group.
- **std()**: Calculates the standard deviation of the values in each group.
- **median()**: Computes the median of the values in each group.

Example:

For example, df is a DataFrame that has the following columns and rows:

Category	Value
A	10
A	20
B	30
B	40

Using the functions sum(), mean() or count() provides a summary of the data:

```
df.groupby('Category')['Value'].sum()
```

Output:

Category	Value
A	30
B	70

```
df.groupby('Category')['Value'].mean()
```

Output:

Category	Value
A	15.0
B	35.0

```
df.groupby('Category')['Value'].count()
```

Output:

Category	Value
A	2
B	2

Custom Aggregation Functions:

You can also write custom functions to handle grouping a set of data. You can use this when you require more advanced operations. It is possible to develop your own custom function and plug it into agg().

If you're looking for the **sum** and **maximum** value for each group, you could write it like this.:

```
df.groupby('Category')['Value'].agg([sum, max])
```

Output:

Category sum max

A 30 20

B 70 40

A custom function can also be used as a parameter in the curry function, e.g:

```
df.groupby('Category')['Value'].agg(lambda x: x.max() - x.min())
```

Output:

Category Value

A 10

B 10

Using Multiple Aggregations:

You can use a list of aggregation functions in the agg() method if you wish to combine them. To find the sum, mean and count for every group, you could use this code:

```
df.groupby('Category')['Value'].agg([sum, mean, count])
```

Thanks to this, you can inspect various statistics easily and in one place.

Transformation and Filtration

Using **transformation** and **filtration** in pandas, you can add or remove certain parts of the data when executing groupby. Both

the transform() and filter() functions can be used to modify data in strong and effective ways.

1. Transformation (transform()**):**

Using the **transform()** function, you can add a function to each group and have the **DataFrame keep the same shape**. You can use this for calculations on each group, so the DataFrame structure does not get altered..

Note, the **transform() function matches the index and shape with the original DataFrame**. You can modify each group without mixing the information into a compact summary.

Example:

For this situation, I have the DataFrame df:

Category	Value
A	10
A	20
B	30
B	40

If you wish to **standardize** the values in each category, subtract the mean from each value and divide by the standard deviation:

```
df['Standardized'] = df.groupby('Category')['Value'].transform(lambda x: (x - x.mean()) / x.std())
```

Output:

Category	Value	Standardized
A	10	-1.0
A	20	1.0
B	30	-1.0
B	40	1.0

In this example, the **transform()** function applies the transformation to each group (by category), and the result is a new column

Standardized that has the same number of rows as the original DataFrame, keeping the structure intact.

2. Filtration (filter()**):**

The **filter()** function is used to **exclude entire groups based on a condition**. You can filter groups by applying a condition on some property of the group (like the size or some aggregation result) and remove those groups that don't meet the condition.

With filter(), you can decide to keep or exclude groups based on a custom condition. The result is a **subset of the original DataFrame**, where some groups are excluded.

Example:

Let's say you only want to keep groups where the sum of the 'Value' column is greater than 40:

```
df_filtered = df.groupby('Category').filter(lambda x: x['Value'].sum() > 40)
```

Output:

Category Value

B 30

B 40

In this case, the group 'A' is excluded because the sum of its 'Value' column (10 + 20 = 30) does not exceed 40. Only group 'B' remains, where the sum is greater than 40 (30 + 40 = 70).

Window Functions in Pandas: Rolling and Expanding

Window functions are commonly used in time series analysis, financial data analysis, and scenarios where you need to perform calculations over a specific window of data. In pandas, two main types of window functions are **rolling()** and **expanding()**. Both are extremely useful for performing calculations like moving averages or cumulative operations, but they work in slightly different ways.

1. Rolling Window Functions (rolling())

The **rolling()** function in pandas is used for **moving window calculations**, such as computing **moving averages**, **moving sums**, **moving standard deviations**, etc. A rolling window applies a function over a fixed-size window, moving step-by-step through the data.

How it works:

- You specify the window size (the number of observations to include in each window).
- The function is applied to each window (i.e., the current window of data at each step), and the result is output for each window.
- The window moves one step at a time until the end of the data.

Example:

Let's say you have a time series of data and you want to calculate a **moving average** with a window size of 3:

```
import pandas as pd
# Sample data
data = {'Date': ['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04', '2021-01-05'],
        'Value': [10, 20, 30, 40, 50]}

df = pd.DataFrame(data)
df['Date'] = pd.to_datetime(df['Date'])

# Rolling mean with a window of 3
df['Rolling_Mean'] = df['Value'].rolling(window=3).mean()

print(df)
```

Output:

Date	Value	Rolling_Mean
2021-01-01	10	NaN

Date	Value	Rolling_Mean
2021-01-02	20	NaN
2021-01-03	30	20.0
2021-01-04	40	30.0
2021-01-05	50	40.0

Here, the rolling(window=3) function creates a moving window of 3 elements, and mean() calculates the mean for each window. The first two rows are NaN because there aren't enough previous data points to form a complete window of size 3.

Common Operations with Rolling:

- **Moving average**: df['Value'].rolling(window=3).mean()
- **Moving sum**: df['Value'].rolling(window=3).sum()
- **Moving standard deviation**: df['Value'].rolling(window=3).std()

2. Expanding Window Functions (expanding())

The **expanding()** function, on the other hand, calculates **cumulative** operations. Unlike rolling(), which uses a fixed window, expanding() applies a function over all the data from the start up to the current row, increasing the window size step-by-step.

How it works:

- The window expands as you move through the data. The first element has just one data point, the second has two, the third has three, and so on.
- It's typically used for cumulative calculations like cumulative sums or cumulative averages.

Example:

Suppose you want to calculate the **cumulative sum** of the 'Value' column:

```
df['Cumulative_Sum'] = df['Value'].expanding().sum()

print(df)
```

Output:

Date	Value	Cumulative_Sum
2021-01-01	10	10
2021-01-02	20	30
2021-01-03	30	60
2021-01-04	40	100
2021-01-05	50	150

In this example, `expanding().sum()` calculates the cumulative sum of the `'Value'` column, progressively adding each new value to the running total.

Common Operations with Expanding:

- **Cumulative sum**: `df['Value'].expanding().sum()`
- **Cumulative mean**: `df['Value'].expanding().mean()`
- **Cumulative standard deviation**: `df['Value'].expanding().std()`

Key Differences Between Rolling and Expanding:

- **Rolling**: The window size is fixed, and calculations are performed over a specific subset of data (i.e., the most recent `n` values).
- **Expanding**: The window expands over time, and calculations are performed using all the data from the start up to the current point.

Exponentially Weighted Functions:

Exponentially Weighted Functions, especially the **Exponentially Weighted Moving Average (EWMA)**, are a powerful tool used in time series analysis to smooth out data. The main advantage of using an exponentially weighted function is that it gives more importance (or weight) to recent observations while giving progressively less weight to older ones. This makes it highly useful

for detecting trends in data where more recent values are considered more relevant than older ones.

The pandas library provides an ewm()function that makes it easy to apply an EWMA in Python. This function applies a window to the input and computes an average that decreases in importance exponentially. In df.ewm(span=..., adjust=True).mean(), the weighting effect of each value in DataFrame df is reported and the span determines how fast the values influence the average. The adjust=True method ensures that the weights are standardized in the proper way.

The method is often used in signal processing, forecasting and financial analysis, as a quick response to recent changes in the data is necessary such as for following changes in stock prices or the weather.

```
import pandas as pd
import matplotlib.pyplot as plt

# Sample time series data
data = {
   'Day': pd.date_range(start='2024-01-01', periods=10, freq='D'),
   'Temperature': [30, 32, 35, 33, 36, 38, 37, 39, 40, 42]
}

df = pd.DataFrame(data)
df.set_index('Day', inplace=True)

# Apply Exponentially Weighted Moving Average with span=3
df['EWMA'] = df['Temperature'].ewm(span=3, adjust=True).mean()

# Display the result
print(df)

# Plot original data and EWMA
plt.figure(figsize=(10, 5))
plt.plot(df['Temperature'], label='Original Temperature')
```

```
plt.plot(df['EWMA'], label='EWMA (span=3)', linestyle='--', color='red')
plt.title('Exponentially Weighted Moving Average')
plt.xlabel('Date')
plt.ylabel('Temperature')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

TIME SERIES ANALYSIS

Time Series Basics

Pandas makes it easy to handle and analyze time series data:

DateTime Index:

Transform the column data to DateTime by using pd.to_datetime() and choose it as the index.

```
# Convert to DateTime
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)
```

Resampling:

To change the frequency of your time series data, apply resample().

```
# Resample to monthly
monthly_df = df.resample('M').mean()
```

ADVANCED TIME SERIES OPERATIONS

Shifting and Lagging:

Use the function shift() to get lags of your data and use diff() to calculate the differences.

```
# Shift data
df['lagged'] = df['value'].shift(1)

# Compute differences
df['diff'] = df['value'].diff()
```

Time Zones:

Time zones can be processed using tz_localize() and tz_convert().

```
# Localize and convert time zones
df = df.tz_localize('UTC').tz_convert('US/Eastern')
```

Periods and DateOffsets:

Include currently and offset date values for convenient time-related calculations..

```
# Create periods
df['period'] = df.index.to_period('M')

# DateOffsets
df['next_month'] = df.index + pd.DateOffset(months=1)
```

More information on Time series is included in module 15 of this book.

Optimizing Pandas Performance in Efficient Storage and Processing

When handling large collections of data in pandas, both the efficiency and how much space they use can be important issues. The library includes various tools and techniques to boost I/O, consume less memory and enhance computation. It is divided into three basic sections: efficient data storage, improving the app's speed and monitoring its functionality.

Pandas is able to handle several file formats, but the performance of each depends on the type of file. Selecting the proper file system can improve the performance of reading and writing data, as well as use less space.

1. The difference between CSV, HDF5 and Parquet

- **CSV**: It can be read by humans, but processing huge files takes a lot of time as there is no compression.

- **HDF5**: Suits large datasets where data is grouped in levels, plus it can be compressed and accessed in no particular order.
- **Parquet**: Parquet is a type of columnar file suitable for running analytics. Built to support big data and speedy data lookup on MongoDB.

Example Parquet for large datasets

```
# Save DataFrame to Parquet format
df.to_parquet('data.parquet')

# Load DataFrame from Parquet
df = pd.read_parquet('data.parquet')
```

2. Feather Format

Feather allows for quick data access, making it a good selection for exchange between Python and R projects.

```
# Save to Feather format
df.to_feather('data.feather')

# Load from Feather
df = pd.read_feather('data.feather')
```

Performance Optimization Techniques

1. Vectorization

Using vectors in NumPy and pandas instead of loops allows for **much faster** calculations than Python code.

Inefficient loop:

```
df['new_col'] = [x + y for x, y in zip(df['col1'], df['col2'])]
```

Vectorized alternative:

```
df['new_col'] = df['col1'] + df['col2']
```

Vectorized operations are often **10–100x faster** than loops.

2. Cython and Numba

For custom computations where vectorization isn't enough, use **Cython** or **Numba** to compile Python code into fast machine code.

Numba Example:

```
from numba import jit
import numpy as np
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({'column': np.arange(1000000)})

@jit(nopython=True)
def custom_function(x):
    return x * 2

# Apply Numba-accelerated function
df['new_column'] = custom_function(df['column'].values)
```

jit(nopython=True) *ensures maximum performance by avoiding Python object overhead.*

3. Parallel Processing

If you're performing tasks that can be split into chunks, use **Joblib** or Python's multiprocessing to process data in parallel.

Example with Joblib:

```
from joblib import Parallel, delayed
import pandas as pd
import numpy as np

# Sample DataFrame
df = pd.DataFrame({'column': np.arange(1000000)})

# Define a chunk processing function
def process_chunk(chunk):
    return chunk * 2
```

```
# Apply in parallel using 4 processes
results = Parallel(n_jobs=4)(
   delayed(process_chunk)(chunk) for chunk in np.array_split(df['column'], 4)
)

# Combine results back into a single Series
df['processed'] = pd.concat([pd.Series(r) for r in results], ignore_index=True)
```

Profiling and Monitoring

Identifying bottlenecks is essential for performance tuning.

1. Code Profiling

Use **cProfile** to time parts of your code and spot inefficiencies.

```
import cProfile

# Profile a groupby operation
cProfile.run("df.groupby('column').mean()")
```

2. Memory Usage Tracking

Get insights into memory consumption with `memory_usage()` and `info()`.

```
# Memory used by each column (deep for object types)
print(df.memory_usage(deep=True))

# Overall DataFrame info
print(df.info())
```

Tip: Reduce memory usage by converting `object` *columns to* `category` *where possible.*

```
df['category_column'] = df['category_column'].astype('category')
```

CASE STUDIES AND PRACTICAL APPLICATIONS

Real-world Data Manipulation

Case Study 1: Clean and transform a large, messy dataset using advanced Pandas techniques.

```
# Load data
df = pd.read_csv('messy_data.csv')

# Clean data
df['date'] = pd.to_datetime(df['date'], errors='coerce')
df['value'] = pd.to_numeric(df['value'], errors='coerce')
df.dropna(inplace=True)

# Transform data
df['log_value'] = np.log(df['value'])
```

Case Study 2: Perform advanced group operations on a real-world dataset to derive insights.

```
# Group by category and calculate summary statistics
grouped = df.groupby('category')
summary = grouped.agg({'value': ['mean', 'std', 'count']})
print(summary)
```

Time Series Forecasting

Case Study 3: Build a time series forecasting model using Pandas and Statsmodels.

```
import statsmodels.api as sm

# Prepare data
df.set_index('date', inplace=True)
df = df.asfreq('D')

# Fit ARIMA model
model = sm.tsa.ARIMA(df['value'], order=(1, 1, 1))
results = model.fit()
print(results.summary())
```

Case Study 4: Analyze and visualize time series data to identify trends and patterns.

```
import matplotlib.pyplot as plt

# Plot time series
df['value'].plot()
```

```
plt.title('Time Series Data')
plt.show()
```

Case study 4: Build a data pipeline that cleans, transforms, and analyzes a large dataset.

```
# Load data
df = pd.read_csv('large_dataset.csv')

# Clean data
df['date'] = pd.to_datetime(df['date'], errors='coerce')
df['value'] = pd.to_numeric(df['value'], errors='coerce')
df.dropna(inplace=True)

# Transform data
df['log_value'] = np.log(df['value'])

# Analyze data
summary = df.groupby('category').agg({'value': ['mean', 'std', 'count']})
print(summary)
```

Case study 5: Create a time series forecasting model using Pandas and additional libraries like Statsmodels or Prophet.

```
import statsmodels.api as sm

# Prepare data
df.set_index('date', inplace=True)
df = df.asfreq('D')

# Fit ARIMA model
model = sm.tsa.ARIMA(df['value'], order=(1, 1, 1))
results = model.fit()
print(results.summary())
```

QUESTIONS

1. Differentiate between **loc[]** and **iloc[]** in Pandas.
2. Perform time series resampling on a dataset with daily data.
3. What is chunking in Pandas, and why is it useful for large datasets?
   Provide an example of reading a CSV file in chunks.
4. What are some techniques to optimize memory usage in Pandas?
   Provide an example of converting a column to a categorical data type.
5. How can you handle missing data in a DataFrame using interpolation?
   Write a code snippet to interpolate missing values in a column.
6. What is the purpose of pd.melt()?
   Provide an example of reshaping a DataFrame using pd.melt().
7. What is a MultiIndex in Pandas?
   Create a MultiIndex DataFrame with at least three levels and perform indexing.
6. How do you access data from a MultiIndex DataFrame using pd.IndexSlice?
   Provide an example of slicing a MultiIndex DataFrame.
7. What is boolean indexing, and how is it used in Pandas?
   Write a code snippet to filter rows where a column's value is greater than 50.
8. How does the query() method improve readability in filtering data?
   Rewrite the following boolean indexing using the query() method:
   `df[df['value'] > 20]`
9. What is the purpose of pd.DataFrame.xs()?

Provide an example of extracting a cross-section from a MultiIndex DataFrame.

10. What is the purpose of the groupby() function in Pandas?
Write a code snippet to group a DataFrame by a column and calculate the mean of another column.
11. How do you apply a custom aggregation function to a grouped DataFrame?
Provide an example of calculating the range (max - min) for each group.
12. What is the difference between transform() and filter() in group operations?
Write a code snippet to apply a transformation and a filter to a grouped DataFrame.
13. What is a rolling window in Pandas?
Provide an example of calculating a 3-day rolling mean for a time series column.
14. How do you calculate an exponentially weighted moving average in Pandas?
Write a code snippet to compute the EWMA for a column.
15. How do you convert a column to a DateTime index in Pandas?
Provide an example of converting a column and setting it as the index.
16. What is resampling in time series analysis?
Write a code snippet to downsample daily data to monthly data and calculate the mean.
17. How do you create a lagged version of a time series column?
Provide an example of shifting a column by 1 period.
18. What is the purpose of tz_localize() and tz_convert()?
Write a code snippet to convert a DateTime index from UTC to US/Eastern time.

19. How do you calculate the difference between consecutive values in a time series?
    Provide an example using the diff() *function.*
20. How can you profile the performance of a Pandas operation?
    Provide an example of using cProfile to profile a groupby operation.
21. What is the purpose of the memory_usage() function?
    Write a code snippet to check the memory usage of a DataFrame.
22. How would you build a time series forecasting model using Pandas and
    Statsmodels?
    Provide a code snippet to fit an ARIMA model to a time series dataset.
23. How do you visualize trends in time series data using Pandas and Matplotlib?
    Write a code snippet to plot a time series column.

MODULE 13

ERRORS AND EXCEPTION HANDLING

Errors and exceptions are inevitable in programming, and handling them properly is crucial for writing robust, maintainable, and error-resilient code. This module covers the fundamentals of errors and exception handling in Python, their importance in data science, and best practices for writing reliable programs.

By the end of this module, the reader will be able to:

1. Understand the Basics of Errors and Exceptions
2. Use Basic Exception Handling Techniques
3. Implement Advanced Exception Handling
4. Raise and Customize Exceptions
5. Apply Exception Handling in Data Science Workflows
6. Debug and Log Exceptions
7. Follow Best Practices for Exception Handling
8. Apply Exception Handling in Practical Data Science Scenarios
9. Write Robust and Error-Resilient Code
10. Understand the Role of Exception Handling in Production Environments
    - Learn how exception handling contributes to the reliability and maintainability of data science applications in production.
    - Implement exception handling strategies for **scalable** and **distributed** data processing systems.

INTRODUCTION

Errors and exceptions are inevitable when writing Python programs, especially in data science, where we work with various data formats, libraries, and complex computations. Proper handling of exceptions ensures that programs run smoothly without unexpected crashes in Python.

BASICS OF ERRORS AND EXCEPTIONS IN PYTHON

In programming, encountering errors is almost as certain as Lagos traffic. Whether you're a beginner or a seasoned developer, your code will run into problems from time to time. But like every Nigerian who knows how to improvise when faced with a challenge, Python also gives us smart ways to handle errors gracefully.

Errors in Python are not something to fear, they're opportunities to write smarter, more resilient code. Whether you're building a financial dashboard or a school grading system, knowing how to detect and handle errors will make your applications more dependable.

TYPES OF ERRORS IN PYTHON

Python errors generally fall into three broad categories, but we'll focus on the two most common ones: **Syntax Errors** and **Runtime Errors**.

Syntax Errors: The Grammar Mistakes of Code

A syntax error occurs when your code breaks the rules of Python's language. Think of it as submitting a WAEC exam in all capital letters and forgetting punctuation. Python immediately flags this and refuses to run your program.

Example:

```
print("Hello World"  # Missing closing parenthesis
```

Output:

SyntaxError: unexpected EOF while parsing

Python reads your code like a meticulous English teacher,once it notices a misplaced comma or an unclosed bracket, it halts everything. These types of errors are caught before the program even begins to run.

Runtime Errors: When Trouble Starts Midway

A runtime error, on the other hand, sneaks in *after* your code has passed the initial check. Imagine you've written a brilliant exam, but forgot to sign your name. It's only during marking that the omission becomes a problem. That's what runtime errors feel like.

Example:

```
print(10 / 0)
```

Output:

ZeroDivisionError: division by zero

Dividing by zero is mathematically undefined, and Python will not allow it. When this happens, the program crashes unless you've told it what to do in such a scenario.

HANDLING RUNTIME ERRORS: THE TRY-EXCEPT BLOCK

Just as Nigerians prepare for rain with umbrellas and generators for power outages, programmers can prepare for errors using **exception handling**. Python provides the try-except block to catch errors and respond sensibly instead of crashing.

Example:

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Output:

Cannot divide by zero!

Instead of an error message stopping your program, Python smoothly prints out your custom message and moves on. This is particularly useful in real-world applications like mobile apps or automated systems, where you want your software to keep running even if a small issue occurs.

Python built-in exceptions

In the journey of writing Python code, built-in exceptions are like road signs on Nigerian highways, ometimes frustrating, but alway s there to guide you when something goes wrong. Python raises th ese exceptions to alert you when you've done something that's log ically or syntactically incorrect, even if your intentions were good. Let's walk through some of the most common ones, with example s to show how and why they occur.

There are various types of built-in python exception

- **ValueError**: This error is raised when a function receives an argument of the correct type but inappropriate value.

  Example:

  ```
  int("abc")  # Cannot convert 'abc' to an integer
  ```

  Output:

  ValueError: invalid literal for int() with base 10: 'abc'

- **TypeError**: Such error is raised when an operation is performed on an object of an inappropriate type.

  Example:

  ```
  "5" + 5  # Cannot concatenate str and int
  ```

  Output:

  TypeError: can only concatenate str (not "int") to str

- **IndexError**: Such error is raised when trying to access an index that does not exist.

  Example:

  ```
  lst = [1, 2, 3]
  ```

```
print(lst[5])  # Index 5 does not exist
```

Output:

IndexError: list index out of range

- **KeyError**: Raised when a dictionary key is not found.

  Example:

```
d = {"name": "Alice"}
print(d["age"])  # Key 'age' does not exist
```

  Output:

  KeyError: 'age'

- **FileNotFoundError**: Raised when a file or directory is not found.

  Example:

```
open("nonexistent_file.txt")  # File does not exist
```

  Output:

  FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent_file.txt'

IMPORTANCE OF EXCEPTION HANDLING IN DATA SCIENCE WORKFLOWS

In the world of data science, where you're often handling massive datasets, unpredictable inputs, and complex pipelines,**exception handling** is not just useful; it is absolutely critical. Imagine you're running a long ETL (Extract, Transform, Load) job overnight, only for it to crash halfway because of a single missing value. That's the kind of situation proper exception handling is designed to prevent.

1. Data Integrity

In data science, messy data is the norm, not the exception. You'll encounter missing values, wrong formats, corrupted files, or even unexpected data types. Without exception handling, these issues can crash your program and disrupt the entire analysis. With it, you can validate and catch errors gracefully:

```
try:
    df = pd.read_csv("data.csv")
except FileNotFoundError:
    print("The dataset could not be found.")
```

2. Robust Pipelines

In production-grade pipelines (e.g., model training, data preprocessing, or feature engineering), one bad input can cause the entire flow to fail. Exception handling helps isolate errors so you can skip or log faulty data while the rest of the pipeline keeps running like a Danfo bus that never stops:

```
for file in file_list:
    try:
        data = pd.read_csv(file)
        # Process data
    except Exception as e:
        print(f"Error processing {file}: {e}")
        continue
```

This ensures your machine learning jobs don't stop halfway because of a single bad file or a NaN where you expected a float.

3. User Experience

When building data science tools or applications, good exception handling leads to clearer, more helpful error messages. Whether it's a data analyst running a script or a manager using your dashboard, giving them useful feedback instead of cryptic Python tracebacks is key:

```
try:
    result = model.predict(user_input)
except ValueError:
    print("Invalid input. Please provide numerical values.")
```

t’s much clearer for users to see a traffic light than just to hear someone yelling “Stop!”.

BASIC EXCEPTION HANDLING TECHNIQUES

Errors are likely to happen in programming, especially in data science. Unlike when your program crashes like an unsecured server, Python assists you with safe and reliable ways to address traffic issues.

Make use of try and except blocks in your code

The primary way Python helps handle errors is the try-except block. You can execute the block of code in an ide and it will intercept certain errors to keep everything working.

Example:

```
try:
   result = 10 / 0
except ZeroDivisionError:
   print("Cannot divide by zero!")
```

Output:

Cannot divide by zero!

Handling Specific Exceptions

Specific kinds of errors can be handled with the help of custom except clauses. Because of this, your code is easier to read and understand.

Example

```
try:
   int("abc")
except ValueError:
   print("Invalid value for conversion!")
```

Output:

Invalid value for conversion!

Multiple Exception Blocks

Dealing with data from users or files from outside can lead to various problems. Many types of exceptions can be managed in Python using several except blocks..

```
try:
   num = int(input("Enter a number: "))
   print(10 / num)
except ZeroDivisionError:
   print("Error: Cannot divide by zero!")
except ValueError:
   print("Error: Invalid input, please enter a number.")
```

Use the else Clause

If no problems arise in the try block in Python, the else clause is carried out. It allows you to keep faulty code out of code designed for when the application is error-free.

Example:

```
try:
   result = 10 / 2
except ZeroDivisionError:
   print("Division by zero!")
else:
   print("Result:", result)
```

Output:

```
Result: 5.0
```

Use the finally Clause

The code beneath the finally clause is executed every time the try-except block ends. It guarantees that key actions, including shutting down resources and closing files, are done every time.
In data science, you could work with an open file or database link. In case of an error in processing, the finally block guarantees that every file or connection is closed properly to avoid memory leaks or distorted data.

Execute the tasks you need to finish such as freeing memory, by adding the finally clause to make sure your code can hold up even during errors.

Example

Because of this example, the file will always close, even if there is an error while reading its contents.

```
try:
   file = open("data.txt", "r")  # Attempt to open the file
   content = file.read()
   print(content)  # Process file content
except FileNotFoundError:
   print("Error: The file was not found.")
finally:
   print("Closing the file...")
   file.close()  # Ensures the file is closed no matter what
```

In the above manner, if the file does not exist within the try block, the except block notices it and handles the FileNotFoundError exception. Whether there is an error or not, in the end, the finally block ensures the code closes the file correctly.

Example:

```
try:
   result = 10 / 0
except ZeroDivisionError:
   print("Division by zero!")
finally:
   print("Execution complete.")
```

Output:

```
Division by zero!
Execution complete.
```

Using else and finally

If no exceptions take place within the try block, the else part will be executed. Doing this allows for easier separation of ordinary program flow from handling errors.
Even if an exception occurs, the finally clause is still run. It is generally employed when ending tasks such as closing files, letting go of resources and terminating database connections, so that important operations still take place when errors are encountered.
Both else and finally support in making exception-handling code effectively structured.

- *else: Runs when no exception occurs.*
- *finally: Executes regardless of an exception.*

Example

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully.")
finally:
    if 'file' in locals():
        file.close()
```

Exceptions that occur in nested coding

In Python, an exception may appear at any point within loops, function calls or while accessing files. It is necessary to use **nested**

try-except blocks to catch errors at the right stage, while avoiding issues that could shut down the program.

Why Use Nested Exception Handling?

- Enables managing errors throughout the different processes of executing a program.
- Assures that small bugs within the system do not end up shutting down an entire program.
- Puts administrators in a better position to resolve errors.
- **Example**:

```
try:
   try:
      result = 10 / 0
   except ZeroDivisionError:
      print("Inner: Division by zero!")
except Exception as e:
   print("Outer:", e)
```

Output:
Inner: Division by zero!

RAISE AND CUSTOMIZE EXCEPTIONS

Raise Exceptions Using the raise Keyword

We can use the raise keyword to cause an exception when certain situations happen. It is helpful when confirming the input, applying restrictions or ending the process if an important issue arises. Thanks to exceptions, errors are stopped and managed instead of leading to deadlocks.

- **Example**:

```
if not isinstance(5, str):
   raise TypeError("Expected a string!")
```

Output:
TypeError: Expected a string!

Create and Use Custom Exception Classes

Custom errors can be created in Python by inheriting from the Exception class. Thanks to custom exceptions, developers can identify various types of errors by looking at the error messages. Distinguishing between errors is most important in huge applications.

- **Example:**

```
class InvalidDataError(Exception):
    pass

def process_data(data):
    if not data:
        raise InvalidDataError("Data cannot be empty!")
    return data

try:
    process_data([])
except InvalidDataError as e:
    print(e)
```

Output:
Data cannot be empty!

APPLICATION OF EXCEPTION HANDLING IN DATA SCIENCE WORKFLOWS

Handle Missing or Corrupted Data

Inaccurate or wrong conclusions can result from missing or corrupt data in data science. Exception handling stops missing values from interrupting the data pipelines. By handling values properly, issues like NaN appearing in unexpected places will be avoided.

- **Example:**

```
data = [1, 2, None, 4]
cleaned_data = []
for value in data:
```

```
    try:
        cleaned_data.append(float(value))
    except (TypeError, ValueError):
        print(f"Skipping invalid value: {value}")
print(cleaned_data)
```

Output:
Skipping invalid value: None
[1.0, 2.0, 4.0]

Safely Read and Write Files

Handling and working with files may result in various errors such as a file that cannot be accessed, denied permission or data that has been altered. Exception handling allows the program to keep track of files, double-check certain aspects and choose an alternative solution when something goes wrong.

- **Example**:

```
try:
    with open("data.csv", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
```

Manage Exceptions During API Calls

API calls can fail due to network issues, incorrect responses, or authentication errors. Exception handling ensures that API requests are retried, fallbacks are used, or informative error messages are logged when failures occur, preventing disruptions in data retrieval processes.

- **Example**:

```
import requests
try:
    response = requests.get("https://api.example.com/data")
    response.raise_for_status()  # Raises HTTPError for bad responses
```

```
except requests.exceptions.RequestException as e:
    print(f"API request failed: {e}")
```

Handle Exceptions During Database Operations

Database operations may fail due to connection errors, incorrect SQL queries, or data integrity violations. Exception handling ensures that database transactions are properly rolled back, reconnections are attempted, and meaningful errors are logged to avoid data corruption or loss.

- **Example**:

```
import sqlite3
try:
    conn = sqlite3.connect("example.db")
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM non_existent_table")
except sqlite3.OperationalError as e:
    print(f"Database error: {e}")
finally:
    conn.close()
```

DEBUG AND LOG EXCEPTIONS

Read and Interpret Tracebacks

Tracebacks provide detailed information about where an error occurred, including the exact line of code and the type of exception. Understanding tracebacks helps in debugging issues efficiently and identifying the root cause of an error.

- **Example**:

```
def faulty_function():
    return 10 / 0

faulty_function()
```

Output:

```
ZeroDivisionError: division by zero
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in faulty_function
```

Python's logging Module

The logging module allows developers to record errors, warnings, and other diagnostic messages. Logging exceptions instead of printing them ensures that error information is stored for debugging, monitoring, and performance analysis in production environments.

- **Example**:

```
import logging
logging.basicConfig(filename="app.log", level=logging.ERROR)
try:
    result = 10 / 0
except ZeroDivisionError as e:
    logging.error(f"An error occurred: {e}")
```

BEST PRACTICES FOR EXCEPTION HANDLING

Avoid Silent Failures

Silent failures occur when exceptions are caught but not properly handled, leading to unpredictable behavior. Instead of ignoring exceptions, developers should log errors, raise appropriate messages, or take corrective actions to prevent data loss or incorrect outputs.

- **Example**:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    pass  # Silent failure (bad practice)
```

Implement Graceful Degradation

Graceful degradation ensures that when errors occur, the system continues running in a limited or fallback mode rather than completely stopping. For example, if a machine learning model

fails to load, a simpler default model can be used instead of terminating the program.

- **Example**:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    result = 0  # Fallback value
```

Provide User-Friendly Error Messages

Technical error messages can confuse end-users. Instead, clear and actionable error messages should be provided to help users understand the issue and take appropriate actions without needing deep technical knowledge.

- **Example**:

```
try:
    int("abc")
except ValueError:
    print("Please enter a valid number!")
```

Write Unit Tests for Exception Handling

Unit tests should include test cases that simulate errors to ensure that exceptions are handled correctly. Testing exception handling ensures that the system behaves as expected even under failure conditions.

- **Example**:

```
import unittest
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b

class TestDivision(unittest.TestCase):
    def test_divide_by_zero(self):
```

```
        with self.assertRaises(ValueError):
            divide(10, 0)

unittest.main()
```

LEVERAGING EXCEPTION HANDLING IN PRACTICAL DATA SCIENCE SCENARIOS

Handle Exceptions During Data Cleaning

Data cleaning involves handling missing values, incorrect data formats, and outliers. Exception handling ensures that errors in data do not cause pipeline failures but are handled properly through data imputation, transformations, or filtering.

- **Example**:

```
import pandas as pd

try:
    df = pd.read_csv("data.csv")
    df["column"] = pd.to_numeric(df["column"], errors="coerce")
except FileNotFoundError:
    print("File not found!")
```

Manage Errors During Model Training

Machine learning models may fail due to issues like insufficient data, incorrect hyperparameters, or convergence problems. Exception handling allows you to identify errors and respond to them by, for example, adjusting the factors or retrying using other data.

- **Example**:

```
from sklearn.linear_model import LinearRegression

try:
    model = LinearRegression()
    model.fit(X_train, y_train)
except ValueError as e:
```

```
    print(f"Model training failed: {e}")
```

Handle Exceptions During Data Visualization

It is possible for data visualisation errors to happen because there is a missing column, the data format is not accepted, or some aspect of rendering cannot be done. If there is an error, exception handling ensures the system offers a new view or shows a useful message.

- **Example**:

```
import matplotlib.pyplot as plt

try:
    plt.plot([1, 2, 3], [4, 5, None])
except ValueError:
    print("Invalid data for plotting!")
```

WRITING ROBUST AND ERROR-RESILIENT CODE

Design Programs to Handle Unexpected Situations

A good program should be ready for errors so that these things do not cause the application to crash. Error handling, safeguards and validations upgrade the strength of the code we write.

- **Example**:

```
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return 0
```

Ensure Smooth Data Science Pipelines

There are numerous steps from collecting data to using it in a model and making it operational. With exception handling in place, any errors encountered during a step will not harm the system's overall performance.

- **Example:**

```
try:
    data = load_data("data.csv")
    cleaned_data = clean_data(data)
    model = train_model(cleaned_data)
except Exception as e:
    print(f"Pipeline failed: {e}")
```

THE ROLE OF EXCEPTION HANDLING IN PRODUCTION ENVIRONMENTS

Improve Reliability and Maintainability

If errors or mistakes go unnoticed in production systems, it could result in the system failing, producing wrong results or becoming vulnerable. With correct exception handling in place, errors are recorded, checked and handled properly, leading to better maintenance of the system.

- **Example:**

```
try:
    process_data(data)
except Exception as e:
    logging.error(f"Error in production: {e}")
    notify_admin(e)  # Send alert to admin
```

Implement Strategies for Scalable Systems

Scalable systems are expected to address errors successfully in distributed systems, cloud environments and when using parallel methods. Tools such as failover, retries and monitoring are used to keep large data processing systems from failing when problems occur.

- **Example:**

```
try:
    result = distributed_computation(data)
```

```
except TimeoutError:
    retry_computation(data)  # Retry on failure
```

When building Python programs for data science, handling exceptions is very important as data, files, APIs and databases are key elements. Following best practices in exception handling ensures that applications are resilient, user-friendly, and maintainable in both development and production environments. 🚀

QUESTIONS

1. What is the difference between a syntax error and a runtime error? Provide an example of each.
2. What is an exception in Python? How is it different from a runtime error?
3. List three common built-in exceptions in Python and explain when they are raised.
4. Why is exception handling important in data science workflows? Provide a real-world scenario.
5. Write a Python code snippet that uses a try and except block to handle a ZeroDivisionError.
6. What is the purpose of handling specific exceptions instead of catching all exceptions generically?
7. Write a Python code snippet that uses multiple except blocks to handle both ValueError and TypeError.
8. What is the purpose of the else clause in a try-except block? Provide an example.
9. Explain the role of the finally clause in exception handling. Write a code snippet to demonstrate its use.
10. What are nested try-except blocks? Provide an example where they might be useful.
11. Can the finally clause be used without an except block? Explain with an example.
12. Why would you want to create a custom exception instead of using a built-in exception?
13. Write a Python function that raises a custom exception if the input is negative.
14. How would you handle missing or corrupted data in a dataset using exception handling? Provide an example.

15. Write a Python code snippet to safely read a file and handle the FileNotFoundError exception.
16. What is a traceback in Python? How can you use it to debug errors?
17. Write a Python code snippet that logs an exception using the logging module.
18. Write a Python code snippet that provides a user-friendly error message for a KeyError.
19. Why is it important to write unit tests for exception handling? Provide an example.
20. What are some common errors that can occur during model training, and how would you handle them?
21. Write a Python function that reads a CSV file, processes its contents, and handles any file-related exceptions.

MODULE 14

PLOTTING AND VISUALIZATION

It is very important we tell stories with our data. Pictures they say speak louder than text, information are better understood when they are visualized. As a datascientist, visualization forms part of your day-to-day tasks.

By the end of this module, students will:

1. Understand the importance of data visualization in data science.
2. Master key Python libraries for visualization (Matplotlib, Seaborn, Pandas, Plotly).
3. Customize and style visualizations for clarity and impact.
4. Create advanced visualizations, including multi-panel figures and interactive dashboards.
5. Apply visualization techniques to real-world datasets and case studies.
6. Tell compelling stories with data using best practices in visualization.
7. Prepare visualizations for reports, presentations, and web applications.
8. Evaluate and critique visualizations for effectiveness and accuracy.

INTRODUCTION TO DATA VISUALIZATION

Making informative visualizations (sometimes called *plots*) is one of the most important tasks in data science. It may be a part of the exploratory process,for example, to help identify outliers or needed data transformations, or as a way of generating ideas for models. For others, building an interactive visualization for the web may be the end goal. Python has many add-on libraries for

making static or dynamic visualizations, but I'll be mainly focused on matplotlib and libraries that build on top of it. matplotlib is a desktop plotting package designed for creating (mostly two dimensional) publication-quality plots.
Over time, matplotlib has spawned a number of add-on toolkits for data visualization that use matplotlib for their underlying plotting. One of these is seaborn, which we will explore in this module.

Why Visualization Matters?

Role in Data Science
Data visualization is a cornerstone of data science, serving as a bridge between raw data and actionable insights. It plays a critical role in two key areas:

1. **Exploratory Data Analysis (EDA)**: Visualization helps data scientists understand the structure, patterns, and anomalies in data. For example, a histogram can reveal the distribution of a variable, while a scatter plot can highlight relationships between two variables.
2. **Communicating Insights**: Visualizations are essential for presenting findings to stakeholders, whether they are technical experts or non-technical decision-makers. A well-designed chart or graph can convey complex information quickly and effectively.

Human Perception
Humans are inherently visual creatures. Research shows that the brain processes visual information 60,000 times faster than text. Visualizations leverage this by:

- **Simplifying Complexity**: A single chart can summarize thousands of data points, making it easier to identify trends, outliers, and patterns.

- **Enhancing Memory**: Visuals are more memorable than numbers or text, ensuring that key insights stick with the audience.
- **Facilitating Decision-Making**: Visualizations enable faster and more informed decisions by presenting data in a digestible format.

Examples of Visualization Impact

- **COVID-19 Dashboards**: During the pandemic, interactive dashboards (e.g., Johns Hopkins University's COVID-19 map) provided real-time updates on cases, deaths, and recoveries. These visualizations helped governments and individuals make informed decisions.
- **Sales Trend Analysis**: Retail companies use line charts to track sales over time, identifying seasonal trends and optimizing inventory management.
- **Healthcare**: Heatmaps are used to visualize patient wait times in hospitals, helping administrators allocate resources more efficiently.

Let's make simple plot

we can try creating a simple plot.

```
In [12]: import numpy as np
In [13]: data = np.arange(10)
In [14]: data
```

```
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [15]: plt.plot(data)
```

Types of Data and Visualizations

Different types of data require different visualization techniques. Below is a breakdown of common data types and the visualizations best suited for them:

Numerical Data

Numerical data consists of quantitative values that can be measured. Examples include age, income, and temperature.

- **Histograms**: Show the distribution of a single numerical variable.

```
import matplotlib.pyplot as plt
plt.hist(df['age'], bins=10, color='blue')
plt.title('Age Distribution')
plt.show()
```

- **Scatter Plots**: Display the relationship between two numerical variables.

```
plt.scatter(df['income'], df['spending'])
plt.title('Income vs Spending')
plt.show()
```

- **Box Plots**: Summarize the distribution of a numerical variable, highlighting the median, quartiles, and outliers.

```
sns.boxplot(x=df['age'])
plt.title('Age Distribution')
plt.show()
```

We will talk more on Numerical data in details later in this modul e.

Categorical Data

Categorical data represents discrete groups or categories. Examples include gender, product categories, and customer segments.

- **Bar Charts**: Compare the frequency or magnitude of different categories.

```
df['category'].value_counts().plot(kind='bar')
plt.title('Product Categories')
plt.show()
```

- **Pie Charts**: Show the proportion of each category in a whole.

```
df['category'].value_counts().plot(kind='pie', autopct='%1.1f%%')
plt.title('Product Categories')
```

```
plt.show()
```

- **Count Plots**: Display the count of observations in each category.

```
sns.countplot(x='category', data=df)
plt.title('Count of Categories')
plt.show()
```

Time-Series Data

Time-series data is collected over time, such as daily stock prices or monthly sales.

- **Line Charts**: Show trends over time.

```
plt.plot(df['date'], df['sales'])
plt.title('Monthly Sales')
plt.show()
```

- **Area Charts**: Similar to line charts but with the area below the line filled, emphasizing volume.

```
plt.fill_between(df['date'], df['sales'], color='skyblue')
plt.title('Monthly Sales')
plt.show()
```

Geospatial Data

Geospatial data includes location-based information, such as city populations or regional sales.

- **Maps**: Visualize data on a geographic map.

```
import geopandas as gpd
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
world.plot()
```

- **Choropleth Maps**: Use color gradients to represent data values across regions.

```
world['population_density'] = world['pop_est'] / world['area']
world.plot(column='population_density', legend=True)
```

Relationships

Visualizations can also show relationships between variables.

- **Heatmaps**: Display correlations or relationships in a matrix format.

```
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

- **Pair Plots**: They illustrate relationships existing between pairs of variables in a dataset.

```
sns.pairplot(df)
plt.show()
```

- **Correlation Matrices**: Summarize relationships between numerical variables.

```
df.corr()
```

We will cover these visualizations in detail later in this module.

GETTING STARTED WITH MATPLOTLIB

Basics of Matplotlib

Most Python programmers use Matplotlib to make static plots. You can use it to create various charts and graphs, like lines and complex charts on several panels. In 2003, John D. Hunter introduced this tool with a user interface like MATLAB and developed it to be highly flexible.

Why Use Matplotlib?

- **Versatile**: Supports line plots, scatter plots, bar charts, histograms, 3D plots, and more.
- **Publication-Quality Output**: Allows fine-tuning of every plot element (fonts, colors, styles).
- **Integration**: Works well with NumPy, Pandas, and Jupyter Notebooks.

- **Cross-Platform**: Compatible with Windows, macOS, and Linux.

Anatomy of a Matplotlib Plot

A Matplotlib plot is made up of various significant parts:

- **Figure**: The main level that assembles all the parts of the plot. It acts as the background for all the plots.
- **Axes**: The area where data is plotted. A figure can contain one or more axes (subplots).
- **Labels**: Descriptive text for the **x-axis**, **y-axis**, and **title** of the plot.
- **Legends**: Keys that explain the symbols, colors, or line styles used in the plot.

Knowing them well helps you customize and improve your visualizations.

1. Figure (plt.figure)

Using plt.figure(), you can create a Figure object to container every chart element in Matplotlib. Your mind can see a figure as an empty canvas that you can use to add one or more plots. Any visual in Matplotlib begins with this statement. You can put various subplots or Axes, side by side or one above the other in the same figure to view and compare them together.

It is possible to customize a figure with the plt.figure() function. For example, you tell Inkscape the size of the figure by using figsize and passing in the values for width and height in inches. This is beneficial when organizing plots for presentations or publications. To adjust the background color of the figure, use the facecolor parameter. Choosing a high dpi ensures the information in your plots remains sharp even when exporting as a file.

In the end, plt.figure() is used to arrange the canvas and its look before you add any subplots or other details. This aspect is critical when making multi-panel plots or enhancing a plot for great look.

Example:

```
fig = plt.figure(figsize=(8, 6), facecolor='lightgray', dpi=100)
```

2. Plotting Axes (also called plt.axes or fig.add_subplot)

The Axes object in Matplotlib is the part of a figure where you display your data. The Figure is the main background and each Axes is a smaller scene within it where different types of charts are shown. With multiple Axes in just a single figure, you can visualize and explain different ideas in the same image.

The Axes object can be made either by specifying its placement with plt.axes() or by structuring subplots with fig.add_subplot(). Axes provide a set of tools and ways to manage various aspects of the plot. For instance, you can use set_xlim() and set_ylim() to highlight only a chosen range of your data. If you want to improve readability in plots that have numbers, set grid(True). Tick marks and their labels can be manipulated with set_xticks() and set_yticklabels() to make the data easier to understand for your viewers.

Example:

```
ax = fig.add_subplot(1, 1, 1)  # 1 row, 1 column, 1st subplot
ax.plot([1, 2, 3], [4, 5, 6])
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Subplot Example')
```

3. Labels such as xlabel, ylabel and title functions

Labels help to make plots more understandable and easier to read. They give viewers details and understanding of each axis and the

entire plot. If a movie is not clearly labeled, its wonderful plot may not make sense to the audience.

Time (s) is added to the X-axis by using the xlabel() function. In the same way, the ylabel() function labels the Y-axis so that the measured variable such as "Temperature (°C)," is easy to understand. They are necessary to indicate what each axis stands for and how the data is represented in those aspects.

It eases communication by specifying a title that gives a quick summary of what the plot shows. Giving the title a clear description helps the viewer notice and remember the main concept of the visualization.

Customization Options:

```
plt.xlabel('X-axis', fontsize=12, color='blue', fontweight='bold')
plt.title('Main Title', fontsize=14, pad=20)  # 'pad' adds space above the plot
```

4. Legend (plt.legend)

To understand the colors, markers and line choices in a Matplotlib plot, a legend is essential. The legend makes it easy for the audience to see at a glance what data points are represented by the various graphical patterns on the chart.

You can call plt.legend() after specifying a label for your line in the main plot. You can call plt.legend() after specifying the labels and it will add a legend to your graph.

The legend allows players to modify it in many ways. You can control its position on the plot using the loc parameter, such as loc='upper left' or loc='best', which automatically selects an optimal location. You can also add a title to the legend using the title parameter for added clarity, and toggle the visibility of the legend frame with frameon=False to create a cleaner look.

Example:

```
plt.plot([1, 2, 3], label='Line 1')
plt.plot([3, 2, 1], label='Line 2')
```

```
plt.legend(loc='best', title='Legend', shadow=True)
```

While libraries like seaborn and pandas's built-in plotting functions will deal with many of the mundane details of making plots, should you wish to customize them beyond the function options provided, you will need to learn a bit about the matplotlib API.

There is not enough room in the book to give a comprehensive treatment to the breadth and depth of functionality in matplotlib. It should be enough to teach you the ropes to get up and running. The matplotlib gallery and documentation are the best resource for learning advanced features.

Creating Basic Plots

Matplotlib supports various types of plots, each suited for different kinds of data representation. Below are some fundamental plot types and how to create them.

1. Line Plot

A **line plot** is a main approach for presenting how values change continuously, for example, during a period of time. Using lines, it links individual points to draw patterns in data that develops in sequence. People often use line plots in time-series research, market trends, record changes in temperature or monitor any type of progress over time.

Creating line plots is quicker in Python because of Matplotlib and Seaborn libraries. In Matplotlib, the plt.plot() function is widely used to change the style and appearance of lines, colors and markers. When you use lineplot() in Seaborn, you can also get confidence intervals for the overall data. You can also enrich your line plot by including notes, making multiple lines visible and even including interactivity tools provided by Plotly.

The line plot should be used mainly to display trends through time or categories that are arranged in a certain order.

```
import matplotlib.pyplot as plt
# Data
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
# Create a line plot
plt.plot(x, y)
# Add labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Simple Line Plot')
# Display the plot
plt.show()
```

Key Customizations:

- Line style: '-' (solid), '--' (dashed), ':' (dotted).
- Color: 'r' (red), 'g' (green), '#1f77b4' (hex code).
- Marker: 'o' (circle), 's' (square), '*' (star).

Example:

```
plt.plot(
   [1, 2, 3, 4],
   [10, 20, 25, 30],
   linestyle='--',
   color='green',
   marker='o',
   label='Trend'
)
plt.legend()
plt.show()
```

Bar Chart

Using a **bar chart** (or bar graph), we can highlight the values of data in categories by making each bar as short or tall as its related value. A bar chart can be used to compare data grouped into sets such as counts, lists of occurrences or metrics that have been brought together from various groups (e.g., sales data for each region, responses to a survey or descriptions of different populations).

It can be used to assess differences in quantities from different categories.

```
import matplotlib.pyplot as plt
# Categories and values
categories = ['A', 'B', 'C']
values = [10, 20, 15]
# Create a bar chart
plt.bar(categories, values)
# Add title
plt.title('Bar Chart')
# Display the plot
plt.show()
```

Types of Bar Charts:

- **Vertical bar chart:** plt.bar(x, height)
  Vertical bar charts (plt.bar()) display categorical data using rectangular bars extending upward from the x-axis. The height of each bar corresponds to the value it represents, making it ideal for comparing discrete categories like monthly sales, survey results, or performance metrics. Vertical bars are best suited when category names are short

and few in number. Customizations like colors, edge styles, and labels help improve readability.

- **Horizontal bar chart:** plt.barh(y, width)
  Horizontal bar charts (plt.barh()) represent data with bars extending horizontally from the y-axis. This format is particularly useful when dealing with long category names or when ranking items (e.g., "Highest to Lowest"). Since text labels appear on the y-axis, they remain legible even with lengthy descriptions. Common use cases include customer satisfaction rankings, inventory comparisons, and demographic distributions.
- **Grouped/Stacked bars:**
  Use width and bottom parameters.
  Grouped (or clustered) bar charts allow side-by-side comparisons of sub-categories within main categories. By adjusting the width parameter and positioning bars with slight offsets, multiple datasets can be visualized together (e.g., sales of Product A vs. Product B across quarters). Grouped bars highlight differences between sub-groups but can become cluttered if too many categories are included.
- Stacked bar charts use the bottom parameter to layer sub-category values on top of one another, showing both individual and cumulative totals. Chart structure can highlight which part of the whole relates to an individual item (such as product revenue and the total amount of revenue). Stacked bars outline the general trends, but it can be tough to compare sub-categories if the segments are very different in size. Icons and colors should be easy to identify through legends.

Example:

```
categories = ['A', 'B', 'C']
values = [15, 20, 12]
```

```
colors = ['red', 'blue', 'green']

plt.bar(categories, values, color=colors, edgecolor='black', linewidth=1.2)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Customized Bar Chart')
plt.show()
```

Scatter Plot

A scatter plot helps to investigate the relationship between two sets of numbers. The use of an x-y axis to mark individual data points unearths correlations, trends, groupings and outliers. An observation is made for every point and its position depends on the values of the variables we are looking at. Therefore, scatter plots are a good choice for finding out if two variables travel together, go in opposite ways or are unrelated.

These plots can be found in statistics, finance and several branches of science. They can show the impact of temperature on how many ice creams sell, as well as the connection between a business's advertising and its income. Apart from correlations, detecting clusters of data points in a scatter plot is important for both segmenting markets and biology investigations. They also mark film records that vary widely from the average, suggesting that there could be errors or unusual developments that need attention.Using Python, you can create scatter plots easily thanks to **Matplotlib, Seaborn and Plotly**. You can increase plot customization with Matplotlib's scatter() function, but with Seaborn's scaterplot() function, you can color by varieties like a category. Plotly's px.scatter() function supports zooming, hovering for information and filtering the underlying data on the fly. Adding regression lines and including labels or marginal histograms helps improve the understanding of the data. Still, working with big datasets may result in scatter plots that are messy, so it helps to use transparency or sampling techniques.

When scatter plots are used correctly, they aid in finding unknown relationships in raw data.

Perfectly suited for seeing the relationship between two numbers.

```
import matplotlib.pyplot as plt
# Data points
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
# Create a scatter plot
plt.scatter(x, y)

# Add title
plt.title('Scatter Plot')

# Display the plot
plt.show()
```

Key Customizations:

- Point size: s=100 (either a scalar or an array).
- Color mapping: **Colormap can be expressed as c=[...] (containing the numbers).**
- Transparency: **alpha=0.5 (0=transparent, 1=opaque).**

```
import matplotlib.pyplot as plt
# Data
x = [1, 2, 3, 4, 5]
y = [10, 15, 12, 18, 20]
sizes = [50, 100, 150, 200, 250]  # Varies point size
colors = [0.1, 0.3, 0.5, 0.7, 0.9]  # Color mapping (numeric values)
alpha = 0.5  # Transparency (0 = transparent, 1 = opaque)
# Create a scatter plot
plt.scatter(x, y, s=sizes, c=colors, alpha=alpha, cmap='viridis')
# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Customized Scatter Plot')
# Show the plot
plt.colorbar()  # Display color bar
plt.show()
```

Customizing Plots in Matplotlib

There are many ways to customize plots with Matplotlib to ensure they look uncluttered and attractive. Altering colors, lines, markers, labels and grid lines gives the visuals a professional effect. Colors can be coded in Chart.js using names, hexadecimal values or RGB tuples and you can even set dashes for type of lines or shape of markers for each data series. Titles, labels for the x and y axis and legends make the plot easier to interpret and the functions help to display text legibly.

Extra changes can be made such as altering figure size, adding gridlines and manipulating the axes limits. To create figures with multiple panels, subplots function is helpful, while annotations provide notes on certain sections. Plot styles can be fixed by using plt.style.use('ggplot' or 'seaborn'). Credits ensure that both information and visual interest are present in the plot of your reports or presentations.

A plot is clear only if the special elements like labels and legends help communicate the information.

- **Titles** (plt.titleThe purpose of the data can be roughly explained by its title (e.g., "Monthly Sales Trends in 2023"). A carefully chosen title can make it clear to the audience what the case is about.
- **Axis Labels** using plt.xlabel() and plt.ylabel() allows you to set the names of each axis and their units such as "Temperature (°C)" or "Revenue (in $). If we do not use labels, the information in the data becomes confusing.
- **Legends** (plt.legend()) you should include legends when you have many lines or groups of data in the same plot. They use various colors/markers for different sets, so it's easy to separate the data.

Example

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

# Plot with labels and legend
plt.plot(x, y, label='Revenue (in $1000s)')  # Assign a label for the legend
plt.xlabel('Quarter')  # X-axis label
plt.ylabel('Sales')    # Y-axis label
plt.title('Quarterly Sales Report')  # Plot title
plt.legend()  # Display legend
plt.show()
```

Table 14-1: Customization Options

Function	Description	Example
plt.title()	Sets the plot title	plt.title('Sales', fontsize=14, color='blue')

Function	Description	Example
plt.xlabel()	Labels the X-axis	plt.xlabel('Time', fontweight='bold')
plt.ylabel()	Labels the Y-axis	plt.ylabel('Temperature (°C)')
plt.legend()	Shows a legend	plt.legend(loc='upper left', shadow=True)

Highlight details in a plot by using different **colors, styles for lines and shapes for points**.

- **Colors** (color parameter):
  Color each dataset differently and check them side by side (you could use 'red' or '#1f77b4' or set them as an RGB tuple). It is possible to distinguish categories with colors or use them to mark out noticeable trends.
- **Line Styles** (known also as linestyle or ls parameter):
  If a trend or hypothetical information exists, show it using solid, dashed, dotted or dash-dot lines.
- **Markers** (marker parameter):
  Symbols such as circles ('o'), squares ('s'), or triangles ('^') are used to mark actual data if your data is sparse or just takes a few values. plt.plot(x, y, marker='s', markersize=8)

Example

```
plt.plot(
   [1, 2, 3, 4],
   [10, 20, 25, 30],
   color='red',        # Line color (name or hex code)
   linestyle='--',     # Line style: '-', '--', ':', '-.'
   marker='o',         # Marker style: 'o', 's', '^', '*'
```

```
    markersize=8,       # Marker size
    markerfacecolor='blue',  # Fill color of marker
    markeredgecolor='black'  # Border color of marker
)
plt.title('Styled Line Plot')
plt.show()
```

Common Customizations

Colors

- Named colors: 'red', 'blue', 'green'
- Hex codes: '#FF5733' (orange), '#1f77b4' (Matplotlib blue)
- Shortcuts: 'r' (red), 'g' (green), 'b' (blue)

Table 14-2: Line Styles

Style	Description
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line

Table 14-3: Markers

Marker	Description
'o'	Circle
's'	Square
'^'	Triangle

Marker	Description
'*'	Star
'D'	Diamond

Setting Axis Limits and Ticks in Matplotlib

Limiting the axes and ticks is necessary to highlight the most important data and increase the clarity of the plot. These functions limit the size of the plot so that unusual points or blank areas do not affect its appearance. If you set plt.ylim(0, 100), the y-axis is limited to values from 0 to 100, so you can see the key trends more clearly.

Otherwise, xticks() and yticks() can calibrate the labels and spaces between them, altering the level of detail (showing every 5 units) or using categories for labels (for example, weekdays). Enhancing clarity on crowded axes can be achieved by fine-tuning the ticks' rotation and font size. Thanks to these tools, the information a plot delivers is clear and not overly complicated.

Example

```
plt.plot([1, 2, 3, 4], [10, 20, 25, 30])
# Set axis limits
plt.xlim(0, 5)  # X-axis range (min, max)
plt.ylim(0, 35)  # Y-axis range
# Customize ticks
plt.xticks([1, 2, 3, 4], ['Q1', 'Q2', 'Q3', 'Q4'])  # Replace numbers with labels
plt.yticks([0, 10, 20, 30], ['0K', '10K', '20K', '30K'])  # Format Y-axis
plt.title('Customized Axes')
plt.show()
```

Table 14-4: **Matplotlib axis customization functions**

Function	**Description**	**Example**
plt.xlim(min, max)	Sets X-axis range	plt.xlim(0, 10)
plt.ylim(min, max)	Sets Y-axis range	plt.ylim(-5, 5)
plt.xticks(ticks, labels)	Customizes X-axis ticks	plt.xticks([1,2,3], ['Jan','Feb','Mar'])
plt.yticks(ticks, labels)	Customizes Y-axis ticks	plt.yticks([0, 50, 100])

Table14-5: Summary Table: Customization Options

Feature	**Function**	**Example**
Title	plt.title()	plt.title('Sales', fontsize=14)
X/Y Labels	plt.xlabel(), plt.ylabel()	plt.xlabel('Time (s)')
Legend	plt.legend()	plt.legend(loc='upper right')
Line Color	color='red'	color='#1f77b4'

Feature	Function	Example
Line Style	linestyle='--'	linestyle=':'
Markers	marker='o'	marker='s'
Axis Limits	plt.xlim(), plt.ylim()	plt.xlim(0, 100)
Ticks	plt.xticks(), plt.yticks()	plt.xticks([1,2], ['Low','High'])

Subplots and Multi-Panel Figures in Matplotlib

In Matplotlib, subplots help you arrange several plots in the same figure. It helps a lot when comparing information from different datasets, exploring several elements of the data or introducing various aspects of one dataset clearly. Rather than drawing several plots, subplots allow you to bring them together next to each other, one above the other or organized in a grid.

One of the most useful ways to plot subplots is by using plt.subplot(nrows, ncols, index) to define how many subplots there are and their order in the chart. Alternatively, to have more options, you can call plt.subplots() and get back a Figure and an array of Axes with which you can plot several subplots and manage the space between them.

Each subplot has its own set of axes, labels and title, making it act as a separate plot. plt.tight_layout() will control the spacing and prevent different parts on the plot from touching. When constructing complex figures, both grid spec and fig.add_subplot() should be handy for styling.

1. Creating Subplots

Basic Subplot Grid

The easiest way to generate subplots is by utilizing plt.subplots() and its result of a figure and an array of axes objects.

```
import matplotlib.pyplot as plt
import numpy as np
# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 8))
# Plot different charts in each subplot
axes[0, 0].plot([1, 2, 3, 4], [1, 4, 9, 16])  # Top-left: Line plot
axes[0, 1].bar(['A', 'B', 'C'], [3, 7, 2])    # Top-right: Bar chart
axes[1, 0].scatter(np.random.rand(10), np.random.rand(10))  # Bottom-left: Scatte
r
axes[1, 1].hist(np.random.randn(1000), bins=20)  # Bottom-right: Histogram

# Add titles to each subplot
axes[0, 0].set_title('Line Plot')
axes[0, 1].set_title('Bar Chart')
axes[1, 0].set_title('Scatter Plot')
axes[1, 1].set_title('Histogram')

# Add a main title for the entire figure
fig.suptitle('Multi-Panel Figure Example', fontsize=16)

plt.tight_layout()
plt.show()
```

Table 14-6: Key Parameters of plt.subplots()

Parameter	Description	Example
nrows	Number of rows	nrows=2
ncols	Number of columns	ncols=2

Parameter	Description	Example
figsize	Figure dimensions (width, height)	figsize=(10, 8)
sharex	Share X-axis between subplots	sharex=True
sharey	Share Y-axis between subplots	sharey=True

Adjusting Layout and Spacing in Matplotlib: Why Use tight_layout()

When creating **multi-panel figures** with subplots in Matplotlib, it's common to encounter **overlapping elements** such as axis labels, titles, and tick marks, especially when plots are closely packed. This can make the visualization look cluttered and reduce readability.

To address this, Matplotlib provides the tight_layout() function, which **automatically adjusts the spacing and padding** between subplots to minimize overlaps. It guarantees that labels and titles are properly displayed and spaced apart automatically. It can be very handy when you're working with multiple subplots or figures that carry long labels or notes.

Often, tight_layout() creates great figures, but not always, particularly when working with complicated graphs or colorbars. Often, you may require plt.subplots_adjust() to make detailed changes to the spacing between the figures.

In short, tight_layout() allows you to quickly clean up multi-panel figures to make them look neat and easy to read.

```
plt.tight_layout(pad=2.0, w_pad=1.0, h_pad=1.0)
plt.show()
```

Manual Spacing Control in Matplotlib Using the subplots_adjust() method

The function tight_layout() does wonders for changing the space between subplots, but can't always achieve the best results if there are complicated annotations, legends, colorbars or labels present. For these situations, Matplotlib makes it easier using the function plt.subplots_adjust().

Manually control the spacing between subplots by adjusting the margins and padding using the subplots_adjust() function. You have the option to customize the layout by adding parameters like:

- left, right: Control the horizontal margins of the entire figure (values between 0 and 1).
- top, bottom: Adjust the vertical margins.
- wspace: Set the width between columns of subplots.
- hspace: Set the height between rows of subplots.

For example:

```
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1, wspace=0.3, hspace=0.4)
```

This way, you can easily make plots that look professional, thanks to **complete control** over the layout. It is most useful when trying to edit subplot displays for materials such as reports, slide decks or dashboards since spacing must be precise.

Example:

```
fig.subplots_adjust(
   left=0.1,   # Left margin
   right=0.9,   # Right margin
   bottom=0.1,  # Bottom margin
   top=0.9,    # Top margin
   wspace=0.4,  # Horizontal space between subplots
   hspace=0.4   # Vertical space between subplots
)
```

Advanced Subplot Techniques in Matplotlib: Uneven Subplot Grids with GridSpec

For times when you want to design subplots that are not evenly sized or balanced, GridSpec from Matplotlib is useful. This is useful because GridSpec actually gives you flexibility in each plot's size and position, in contrast to the standard plt.subplot() and plt.subplots() functions that arrange the subplots in nice, equal rows and columns which is good when all the subplots are necessary to your story.

Part of the Matplotlib gridspec module, GridSpec helps you organize a layout (e.g., in 3 rows by 3 columns) and then decide the dimensions for each part of the grid.

Example:

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
fig = plt.figure(figsize=(8, 6))
gs = gridspec.GridSpec(3, 3)
# Large plot spanning the top row
ax1 = fig.add_subplot(gs[0, :])
# Two smaller plots on the second row
ax2 = fig.add_subplot(gs[1, :2])
ax3 = fig.add_subplot(gs[1:, 2])

# A wide plot on the bottom-left
ax4 = fig.add_subplot(gs[2, :2])
```

In this layout created using Matplotlib's GridSpec, the figure is divided into a 3×3 grid, allowing for a more flexible arrangement of subplots. The first subplot is made to cover all three columns in the top row which allows you to provide a broad or key visual that the following plots can build on. Beneath it, ax2 fills up the first two columns in the second row, offering an extra view for more information. Next, ax3 is positioned in the far right column. It's as wide as the previous two variables but rises much higher, good for metrics, time series or stacked bars. Finally, ax4 takes up the first two slots of the third row that are in the bottom-left corner. Consequently, the way the artwork is made gives each plot a noticeable presence, creating balance and adding depth to the narrative expressed in the figure.

Nested Subplots (Subplot within Subplot)

```
fig = plt.figure(figsize=(8, 6))
# Main axes
ax_main = fig.add_subplot(1, 1, 1)
ax_main.plot([1, 2, 3], [1, 2, 3])

# Inset axes (small plot inside main plot)
ax_inset = fig.add_axes([0.6, 0.6, 0.25, 0.25])  # [x, y, width, height]
ax_inset.hist(np.random.randn(100), bins=15)
ax_inset.set_title('Inset Plot')

plt.show()
```

Table 14-7: Summary Table: Subplot Techniques

Method	Use Case	Example
plt.subplots()	Basic grid of subplots	fig, axs = plt.subplots(2, 2)

Method	Use Case	Example
plt.subplot()	Single subplot (index-based)	plt.subplot(2, 2, 1)
add_subplot()	Add subplots to a figure object	ax = fig.add_subplot(1, 2, 1)
GridSpec	Complex, flexible layouts	gs = GridSpec(2, 2); ax = fig.add_subplot(gs[0, :])
subplots_adjust()	Fine-tune spacing between subplots	plt.subplots_adjust(wspace=0.5, hspace=0.3)

INTRODUCTION TO SEABORN

Seaborn provides an easy-to-use interface for making beautiful and significant statistics plots based on Matplotlib. Seaborn makes Matplotlib better, simplifying coding and offering statistical tools which is why it is preferred for inspecting and analyzing data.

One of the main advantages of Seaborn is that it **integrates with Matplotlib without difficulty**. Because every Seaborn plot is a Matplotlib object, users have the option to alter their plots with

Matplotlib tools once the initial plot is done. Because of this flexibility, it is simple to create prototypes and refine the design.

Additionally, Seaborn lets you use straightforward, easy-to-understand code for making complex graphics. Generating lines for regression, plots for distribution or comparing multiple variables is often possible in Seaborn using a single function, this is in contrast to working with multiple lines in Matplotlib. Been designed intelligently, so that editing a plot with new data is effortless.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Apply default Seaborn theme
sns.set_theme()
```

Default Styles & Color Palettes in Seaborn

With Seaborn, defaults and suitable colors are used to enhance each visualization, making plots attractive and easy to interpret. To improve legibility, the library uses whitegrid or darkgrid themes to place light but helpful grids on the plot.

Seaborn uses colormaps such as viridis and rocket which are perceptually uniform and useful for everyone, whether colorblind or not. They prevent the problem of deceiving appearances by always having colored gradients of the same brightness.

Furthermore, Seaborn sizes all parts of a plot to match each other, so the user does not need to change them manually for different graphics. People can choose a different style with sns.set_style() or change the colors with sns.set_palette(), but the initial settings save most users time and effort for professional results.

Example: Setting a Style

```
sns.set_style("darkgrid")  # Options: white, dark, whitegrid, darkgrid, ticks
```

```
sns.set_palette("husl")    # Set color palette
```

Key Plot Types

1. Distribution Plots

Plotting data by distribution helps in spotting the main features, including how central the data is, how much it is skewed and the values that lie far from the rest. Creating graphs with Seaborn is simple using straightforward functions:

Table 14-8a: **Common Seaborn Distribution Plots**

Plot Type	Function	Use Case	Example
Histogram	sns.histplot()	Frequency distribution	sns.histplot(data=df, x='age', bins=20, kde=True)
KDE Plot	sns.kdeplot()	Smooth density estimate	sns.kdeplot(data=df, x='income', fill=True)
Rug Plot	sns.rugplot()	Marginal distributions	sns.rugplot(data=df, x='age')

Example:

```
# Combined histogram and KDE
sns.histplot(data=df, x='price', kde=True, bins=15)
plt.title("Price Distribution")
plt.show()
```

2. Categorical Plots

With categorical plots, you can examine numbers that belong to different groups and notice any increases, decreases or unusual

values among them. Using Seaborn, you can generate these graphics using only a handful of lines of code and they are easy to adapt.

Table 14-8b: Common Seaborn Distribution Plots – Functions, Use Cases, and Examples

Plot Type	Function	Use Case	Example
Bar Plot	sns.barplot()	Compare means	sns.barplot(x='class', y='survival_rate', data=df)
Box Plot	sns.boxplot()	Show quartiles & outliers	sns.boxplot(x='species', y='petal_length', data=df)
Violin Plot	sns.violinplot()	Distribution + density	sns.violinplot(x='day', y='total_bill', data=df)
Swarm Plot	sns.swarmplot()	Point distribution	sns.swarmplot(x='species', y='sepal_width', data=df)

Example:

```
# Box plot with hue
sns.boxplot(data=df, x='species', y='sepal_length', hue='region')
plt.title("Sepal Length by Species")
plt.show()
```

3. Relationship Plots

Correlation plots are created to show how one numeric variable is connected or depending on other variables. With Seaborn, you can easily build these charts and still support advanced statistical operations.

Table 14-8c: Common Seaborn Distribution Plots – Functions, Use Cases, and Examples

Plot Type	Function	Use Case	Example
Scatter Plot	sns.scatterplot()	2D relationships	sns.scatterplot(x='height', y='weight', data=df)
Line Plot	sns.lineplot()	Trends over time	sns.lineplot(x='year', y='sales', data=df)
Pair Plot	sns.pairplot()	All pairwise relationships	sns.pairplot(df, hue='species')
Heatmap	sns.heatmap()	Correlation matrix	sns.heatmap(df.corr(), annot=True)

Example:

```
# Scatter plot with regression line
sns.lmplot(data=df, x='engine_size', y='mpg', hue='fuel_type')
plt.title("Engine Size vs. MPG")
plt.show()
```

Advanced Features

Faceting (Small Multiples)

A FacetGrid allows seaborn to separate the data into different categories and plot each subset in its own panel. Using this method, patterns are compared well across groups because all axes remain the same.

```
g = sns.FacetGrid(df, col="region", row="gender", height=3)
g.map(sns.scatterplot, "age", "income")
g.add_legend()
```

Example:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Create DataFrame
data = {
   'total_bill': [25, 18, 40, 15, 30, 22, 50, 10, 45, 20],
   'tip': [4, 2, 7, 1, 5, 3, 10, 1, 8, 2],
   'gender': ['Male', 'Female', 'Male', 'Female', 'Male', 'Female', 'Male',
'Female', 'Male', 'Female'],
   'smoker': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'No'],
   'day': ['Sun', 'Sat', 'Sun', 'Sat', 'Sun', 'Sat', 'Sun', 'Sat', 'Sun', 'Sat'],
   'time': ['Dinner', 'Lunch', 'Dinner', 'Lunch', 'Dinner', 'Lunch', 'Dinner',
'Lunch', 'Dinner', 'Lunch']
}
df = pd.DataFrame(data)
print(df.head())
g = sns.FacetGrid(df, col="time", row="smoker", margin_titles=True,
height=4)
g.map(sns.histplot, "total_bill", kde=True, bins=5, color="skyblue")
g.set_axis_labels("Total Bill ($)", "Frequency")
g.set_titles(col_template="{col_name}", row_template="Smoker:
{row_name}")
plt.tight_layout()
plt.show()
```

Custom Themes & Palettes

With Seaborn, the uniformity and beauty of plots can be maintained by using different themes and color palettes. You can change the look of all your plots by customizing parameters such as the background theme, font sizes and grid display with the sns.set_theme() function and by setting color palettes for plots using sns.set_palette(). Tools allow you to maintain your branding, like using striking contrast for presentations and make your designs easy to read for everyone (e.g., colorblind palettes). With the help of sns.color_palette(), individuals can design visuals to suit their specific needs while keeping Seaborn's statistics intact.

Example

```
# Set theme and palette
sns.set_theme(style="ticks", palette="deep", font_scale=1.2)

# Create a custom diverging palette
sns.set_palette(sns.diverging_palette(220, 20, as_cmap=True))
```

Regression Plots

Regression analysis in Seaborn allows the user to check the **association between two continuous variables** and to plot **a fitted line** predicting that relationship. The most popular choice for this job is sns.regplot() which responds by displaying a scatter plot and a linear regression line which is already included. By doing this, we are able to determine if the relationship goes up, down or up and down.

With seaborn, you can also use the function sns.lmplot() which behaves like regplot and allows you to chart data from different subgroups using features such as hue, col or row. In EDA, these plots play a crucial part since they use statistics and visuals to clarify data. You can also personalize seaborn regression plots by

choosing confidence intervals, polynomial regression of the desired order and using robust fitting.

```
sns.regplot(data=df, x='ad_spend', y='sales', ci=95)
import seaborn as sns
import matplotlib.pyplot as plt

# Sample dataset
tips = sns.load_dataset("tips")

# Simple regression plot
sns.regplot(x="total_bill", y="tip", data=tips)
plt.title("Regression Plot: Total Bill vs Tip")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.show()
```

Example 2

```
# Regression plot grouped by 'sex'
sns.lmplot(x="total_bill", y="tip", hue="sex", data=tips)
plt.title("Regression Plot by Gender")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.show()
```

In the first case, the regression line is easy to see, since there are scatter points. In the second case, we see that two different regression lines are created for men and women, letting us catch their individual trends easily.

Quick Visualizations Using Pandas

With Pandas, users can easily plot data from DataFrames and Series using Matplotlib tools. They are most useful for prompt exploratory data analysis since you do not need to use complex libraries like Seaborn or Plotly.

You can make several types of plots using a single line of code. For example, using df['price'].plot(kind='hist', bins=20) produces a histogram that shows how the price information is distributed

among the rows. Similarly, producing a scatter plot with df.plot.scatter(x='age', y='income') can reveal if there are any relationships or unusual points between two continuous variables. Pandas also makes styled DataFrames available such that you can apply a gradient color map to show correlations in df.corr().style.background_gradient(cmap='coolwarm') and see them more clearly.

```
df['price'].plot(kind='hist', bins=20)  # Histogram
df.plot.scatter(x='age', y='income')    # Scatter plot
df.corr().style.background_gradient(cmap='coolwarm')#Styled correlation matri
```

Exploratory Data Analysis (EDA)

Doing **EDA** is very important in all data science or machine learning work and **Seaborn** allows you to perform EDA easily using informative graphics. Meaningfully organized by Seaborn, the distribution, patterns, relationships and structure in a dataset are much easier to interpret than in Matplotlib.

Seaborn presents a wide range of charts suitable for EDA tasks. Thanks to **sns.histplot()** and **sns.boxplot()**, we can easily view the distribution and variance of any given feature and notice when they deviate from normal. For spotting relatedness between features, we can also see many scatter plots at once using **sns.pairplot()** which is built for this purpose. When working with categories, **countplot()** and **barplot()** display the counts and summaries of the values for each category. To summarize, people tend to use **sns.heatmap()** to visualize the relationships between many variables all at once.

The unique look and context options allow Seaborn to make EDA visuals both intuitive and informative. Profiling a new set of data or checking your assumptions before modeling, you can easily look for patterns using Seaborn's tools.

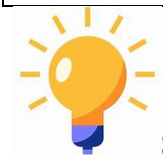
```
# Summary of distributions
```

```
df.hist(figsize=(10, 8), bins=15)

# Pairwise relationships
sns.pairplot(df, hue='target_column')
```

Table 14-9: **Summary Table: Seaborn vs. Pandas Plotting**

Feature	Seaborn	Pandas
Statistical Plots	✔ (Box, Violin, KDE)	✖
Customization	High (themes, palettes)	Low
Integration	Works with DataFrames	Native DataFrame support
Best For	Statistical analysis	Quick EDA



Start with `sns.set_theme()` for better defaults.
Use `hue` **parameter** to encode categorical variables.
Leverage faceting for multi-dimensional analysis.
Combine with Matplotlib for final tweaks (e.g., `plt.title()`).

Interactive and Geospatial Visualization with Plotly & GeoPandas

Interactive Visualizations with Plotly

Plotly is a robust and flexible Python library for creating **interactive, publication-quality visualizations** that can run directly in web browsers or be embedded in Jupyter Notebooks

and dashboards. Unlike static plotting libraries such as Matplotlib, Plotly visualizations allow users to **hover, zoom, pan, and filter data** in real-time, making data exploration much more dynamic and insightful.

Plotly supports a wide range of chart types, including **line plots, bar charts, scatter plots, bubble charts, heatmaps, 3D plots, and maps**. It is built on top of the JavaScript library D3.js and offers two main interfaces in Python: **Plotly Express** (a high-level API for quick plotting) and the **Graph Objects API** (a low-level API for building highly customized visuals).

Plotly is particularly useful in interactive dashboards and web applications using frameworks like Dash. It also integrates well with Pandas, allowing you to pass DataFrames directly into plotting functions, and supports exporting visuals to HTML, PNG, or PDF.

Key Features of Plotly

One thing that makes Plotly unique is its ability to make data visualization interactive and interesting to look at. Listed below are some important features of this site:

- **Hover Tooltips** – This feature in Plotly makes it easy to see precise numbers when you move over data points.
- **Zoom and Pan Interactions** – With these interactions, you can enlarge some parts of a plot or smoothly browse along the axis bars, all without having to recreate the chart.
- **3D Plots and Animations** – With Plotly, you can make 3D plots, animate data over time and observe how data changes over time.
- **Built-in Themes** – Plotly offers a choice of built-in styles and themes (plotly_dark, ggplot2, seaborn, etc.) that quickly give any visualization a great look for publication.

Basic Example

```
import plotly.express as px
# Interactive scatter plot
fig = px.scatter(
   data_frame=df,
   x='GDP_per_capita',
   y='Life_Expectancy',
   color='Continent',
   size='Population',
   hover_name='Country',
   title='Life Expectancy vs. GDP per Capita'
)
fig.show()
```

Advanced Plotly Features

3D Plots

It is easy to view multidimensional data in 3D visualizations using Plotly. With limited lines of code, you can develop 3D scatter plots, surface plots, mesh plots and line plots that can be rotated and zoomed in.

You can use 3D plots to identify patterns and groups of data from three continuous variables. You can find 3D features in Plotly's graph_objects module in go.Scatter3d, go.Surface and go.Mesh3d. The charts allow users to move them, zoom them and hover over various spots to check their data, making them useful for visualization in presentations or on dashboards.

Example: Scatter plots shown using a 3D graph

```
import plotly.graph_objects as go
import pandas as pd

# Sample data
df = pd.DataFrame({
   'x': [1, 2, 3, 4],
   'y': [10, 15, 13, 17],
   'z': [5, 6, 7, 8]
```

```
})

fig = go.Figure(data=[go.Scatter3d(
   x=df['x'],
   y=df['y'],
   z=df['z'],
   mode='markers',
   marker=dict(
      size=8,
      color=df['z'],  # Color by z value
      colorscale='Viridis',
      opacity=0.8
   )
)])

fig.update_layout(title='3D Scatter Plot',
                  scene=dict(
                     xaxis_title='X Axis',
                     yaxis_title='Y Axis',
                     zaxis_title='Z Axis'))

fig.show()
```

Here, Plotly emphasizes the value of interactivity, taking data analysis to a new level by supporting 3D graphs instead of the standard 2D lines.

Example 2:

```
fig = px.scatter_3d(
   df,
   x='Height',
   y='Weight',
   z='Age',
   color='Gender'
)
fig.show()
```

Animations in Plotly

With Plotly, animating charts is not difficult so you can watch data move through time or change depending on a category. Animation can be added to charts by defining a change in frame using the animation_frame parameter. With this feature, visualizations are more vivid and allow viewers to notice developing trends.

Example:Using an Animated Bar Chart

```
import plotly.express as px

# Sample data structure (assumes df has columns 'Year', 'Sales', and 'Quarter')
fig = px.bar(
    df,
    x='Year',
    y='Sales',
    animation_frame='Quarter',
    range_y=[0, 1000]
)
fig.show()
```

This code creates an animated bar chart where each frame represents a different **quarter**, and bars reflect the **sales** values for different years. This is helpful in dashboards or business reports where temporal change is important.

Dash for Dashboards

Dash is a Python framework by Plotly for building **interactive, web-based dashboards** using Python only,no JavaScript required. It uses components like dcc.Graph and dcc.Slider to connect interactive elements to visualizations.

Example: Basic Dash App

```
import dash
from dash import dcc, html
```

```
# Use an existing Plotly figure (e.g., fig from above)
app = dash.Dash(__name__)

app.layout = html.Div([
   dcc.Graph(figure=fig),
   dcc.Slider(min=0, max=10, step=1, value=5)
])

app.run_server(debug=True)
```

This simple app shows an interactive graph with a slider. You can connect the slider to filter or animate plots, making dashboards truly **interactive and user-driven**.

Geospatial Visualization

GeoPandas Basics

GeoPandas is an extension of Pandas that enables you to work with **geospatial data**, such as **shapefiles** or **GeoJSON**. It adds support for geometric operations and plotting maps directly with Matplotlib.

With GeoPandas, you can easily load and visualize global, national, or regional boundaries, and even overlay datasets like population, GDP, or pollution.

Example: Plotting a World Map

```
import geopandas as gpd
import matplotlib.pyplot as plt

# Load built-in world data
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))

# Plot the world map
world.plot(figsize=(10, 6), edgecolor='black')
plt.title('World Map')
plt.show()
```

This example loads a low-resolution shapefile of world countries and renders a clean map. GeoPandas allows further customization

like coloring by country population or overlaying additional spatial layers.

Choropleth Maps

Choropleth maps let you visualize information about locations using a particular color for every region with that characteristic (for example, population density, GDP or unemployment). Such maps help display trends, differences and similarities between various parts of the world.

Making choropleth maps is easy using GeoPandas. Once a column for the value to be visualized is added (like population density), map colors can represent the numbers or figures, making it simpler to read or compare the information.

Example: Choropleth Map Illustrating the Population Density of Each Country

```
import geopandas as gpd
import matplotlib.pyplot as plt

# Load world data
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))

# Calculate population density
world['population_density'] = world['pop_est'] / world['area']

# Plot choropleth map
world.plot(
    column='population_density',
    legend=True,
    cmap='OrRd',  # Color map for visualization
    figsize=(12, 8),
    scheme='quantiles'  # Quantile-based color scheme for classification
)

# Set title
plt.title('Population Density by Country')
```

```
plt.show()
```

Here, we use the given example:

- **Population density** is found by dividing the population figure by the country's size.
- Using the **OrRd color scale**, the map reflects that lower population densities are represented by yellow and higher densities are represented by red..
- The equidistant coloring in coordcat makes each part of the data about the same size, allowing for a more balanced display.

From the choropleth map, the darker shades of red on the map represent countries with larger population densities.

They are valuable for highlighting differences between areas in various data, especially for spotting any noticeable patterns in locations..

Table 14-10:Hands-On Case Studies

Case Study	Tools Used	Key Visualization
Sales Trend Analysis	Matplotlib, Seaborn	Line plot with seasonal decomposition
COVID-19 Dashboard	Plotly, Dash	Interactive time-series + maps
Geospatial Analysis	Geopandas, Folium	Choropleth + point maps
Correlation Analysis	Seaborn	Heatmap + pair plot

Advanced Topics

ML Model Interpretation

It is vital to interpret the actions of complex machine learning models and see which elements are affecting their predictions. A number of approaches and representations are available to help people interpret these models easily:

- **SHAP (Shapley Additive Explanations)**: SHAP credit each part of the input to an answer, helping to understand what a machine learning model focuses on. **shap.plots.waterfall()** displays this data in a row of added predictions compared to the average result.

  Example of a SHAP Waterfall Plot:

```
import shap
import matplotlib.pyplot as plt

# Assuming model and X_test are pre-defined
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)

# SHAP waterfall plot for the first instance
shap.plots.waterfall(shap_values[0])
plt.show()
```

- Looking at this plot, you will notice how each feature affects the model's decision. The effects of single features are shown in sequence, similar to a waterfall, so it's simple to understand their importance for a prediction.
- This function with Sklearn allows you to plot the relationship between one feature and the overall prediction with all other features held constant. It is easy to use PDPs to explore the relationship between a variable and a feature, especially in more complicated models.

Example of a Partial Dependence Plot is provided below:

```
from sklearn.inspection import plot_partial_dependence
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor()
model.fit(X_train, y_train)  # Fit the model

plot_partial_dependence(model, X_train, features=[0, 1], feature_names=['Feature 1', 'Feature 2'])
plt.show()
```

This plot allows us to see how the chosen features impact the model and understand the changes in its overall performance.

Big Data Visualization

Processing and presenting large data sets (millions or billions rows) is not always easy. Despite some slowness in standard libraries, specific tools exist for the smooth and efficient display of data:

- **Datashader**: A Python library that allows you to quickly display huge amounts of points in a plot. It can handle large datasets and produce high-quality, interactive visualizations quickly by intelligently optimizing the display of data points.

  Example (Datashader Scatter Plot):

```
import datashader as ds
from datashader import transfer_functions as tf
import pandas as pd
import numpy as np

# Generate large random data
n = 1000000
df = pd.DataFrame({'x': np.random.randn(n), 'y': np.random.randn(n)})

# Create a canvas and render
canvas = ds.Canvas(plot_width=800, plot_height=600)
```

```
agg = canvas.points(df, 'x', 'y')

# Convert to image
img = tf.shade(agg)
img.show()
```

Datashader efficiently handles large data visualizations by rendering only the necessary data points for display, improving performance while maintaining clarity.

- **Vaex**: A library optimized for **out-of-core DataFrames**, allowing users to manipulate and visualize large datasets that don't fit into memory. Vaex provides fast operations on datasets up to **billions of rows** by using lazy evaluations and memory-mapping techniques.

  Example (Vaex DataFrame and Plotting):

```
import vaex

# Load a large dataset
df = vaex.open('large_dataset.hdf5')

# Perform operations and plot
df['log_value'] = df['value'].log()  # Lazy operation
df.plot(df['log_value'], kind='histogram', bins=50)
```

Vaex allows you to work with massive datasets without running into memory issues, and it efficiently performs data transformations and visualizations.

Table 14-11: Useful Tip

Library	Use Case	Key Function
Plotly	Interactive plots	`px.scatter()`, `px.line()`

Library	Use Case	Key Function
Geopandas	Geospatial maps	gpd.read_file(), .plot()
Dash	Dashboards	dcc.Graph(), html.Div()
Datashader	Big data	ds.Canvas(), tf.shade()

Advanced Figures and Subplots

Plots in matplotlib reside within a Figure object. You can create a new figure with plt.figure:

```
In [16]: fig = plt.figure()
```

In IPython, an empty plot window will appear, but in Jupyter nothing will be shown until we use a few more commands. plt.figure has a number of options; notably, figsize will guarantee the figure has a certain size and aspect ratio if saved to disk.

You can't make a plot with a blank figure. You have to create one or more subplots using add_subplot:

```
In [17]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be 2 × 2 (so up to four plots in total), and we're selecting the first of four subplots (numbered from 1).

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)
In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

One nuance of using Jupyter notebooks is that plots are reset after each cell is evaluated, so for more complex plots you must put all of the plotting commands in a single notebook cell.

Here we run all of these commands in the same cell:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
```

```
ax3 = fig.add_subplot(2, 2, 3)
```

When you issue a plotting command like plt.plot([1.5, 3.5, -2, 1.6]), matplotlib draws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation.

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

The 'k--' is a *style* option instructing matplotlib to plot a black dashed line. The objects returned by fig.add_subplot here are AxesSubplot objects, on which you can directly plot on the other empty subplots by calling each one's instance method

```
n [21]: _ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)
In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

You can find a comprehensive catalog of plot types in the matplotlib documentation.

Creating a figure with a grid of subplots is a very common task, so matplotlib includes a convenience method, plt.subplots that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [24]: fig, axes = plt.subplots(2, 3)
In [25]: axes
```

```
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626374048>,
<matplotlib.axes._subplots.AxesSubplot object at 0x7fb62625db00>,
<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6262f6c88>],
[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6261a36a0>,
<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626181860>,
<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6260fd4e0>]], dtype
=object)
```

This is very useful, as the axes array can be easily indexed like a two-dimensional array; for example, axes[0, 1]. You can also indicate that subplots should have the same x- or y-axis using sharex and sharey, respectively. This is especially useful when

you're comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently. See Table 9-1 for more on this method.

Table 14-12. pyplot.subplots **Options:**

Argument	Description
Nrows	Number of rows of subplots
Ncols	Number of columns of subplots
Sharex	All subplots share the same x-axis ticks (adjusting the xlim affects all subplots)
Sharey	All subplots share the same y-axis ticks (adjusting the ylim affects all subplots)
subplot_kw	Dictionary of keyword arguments passed to the add_subplot call used to create each subplot
**fig_kw	Additional keyword arguments used when creating the figure, e.g., plt.subplots(2, 2, figsize=(8, 6))

Adjusting the spacing around subplots

By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. You can change the spacing using the subplots_adjust method on Figure objects, also available as a top-level function:

subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)

wspace and hspace controls the percent of the figure width and figure height, respectively, to use as spacing between subplots.

Here is a small example where I shrink the spacing all the way to zero:

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
for j in range(2):
axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

You may notice that the axis labels overlap. matplotlib doesn't check whether the labels overlap, so in a case like this you would need to fix the labels yourself by specifying explicit tick locations and tick labels (we'll look at how to do this in the following sections).

Advanced Colors, Markers, and Line Styles

Matplotlib's main plot function accepts arrays of x and y coordinates and optionally a string abbreviation indicating color and line style. For example, to plot x versus y with green dashes, you would execute:

```
ax.plot(x, y, 'g--')
```

This way of specifying both color and line style in a string is provided as a convenience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

```
ax.plot(x, y, linestyle='--', color='g')
```

There are a number of color abbreviations provided for commonly used colors, but you can use any color on the spectrum by specifying its hex code (e.g., '#CECECE').

You can see the full set of line styles by looking at the docstring for plot (use plot? In IPython or Jupyter).

Line plots can additionally have *markers* to highlight the actual data points. Since matplotlib creates a continuous line plot,

interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style:

```
In [30]: from numpy.random import randn
In [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

This could also have been written more explicitly as:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the drawstyle option

```
In [33]: data = np.random.randn(30).cumsum()
In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>]
In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>]
In [36]: plt.legend(loc='best')
```

You may notice output like <matplotlib.lines.Line2D at ...> when you run this. Matplotlib returns objects that reference the plot subcomponent that was just added. A lot of the time you can safely ignore this output. Here, since we passed the label arguments to plot, we are able to create a plot legend to identify each line using plt.legend.

You must call plt.legend (or ax.legend, if you have a reference to the axes) to create the legend, whether or not you passed the label options when plotting the data.

Advanced Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural pyplot interface (i.e., matplotlib.pyplot) and the more object-oriented native matplotlib API.

The pyplot interface, designed for interactive use, consists of methods like xlim,
xticks, and xticklabels. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

- Called with no arguments returns the current parameter value (e.g., plt.xlim() returns the current x-axis plotting range)
- Called with parameters sets the parameter value (e.g., plt.xlim([0, 10]), sets the x-axis range to 0 to 10)

All such methods act on the active or most recently created AxesSubplot. Each of them corresponds to two methods on the subplot object itself; in the case of xlim these are ax.get_xlim and ax.set_xlim. I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk:

```
In [37]: fig = plt.figure()
In [38]: ax = fig.add_subplot(1, 1, 1)
In [39]: ax.plot(np.random.randn(1000).cumsum())
```

To change the x-axis ticks, it's easiest to use set_xticks and set_xticklabels. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using set_xticklabels:

```
In [40]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
In [41]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
....: rotation=30, fontsize='small')
```

The rotation option sets the x tick labels at a 30-degree rotation. Lastly, set_xlabel gives a name to the x-axis and set_title the subplot title

```
In [42]: ax.set_title('My first matplotlib plot')
Out[42]: <matplotlib.text.Text at 0x7fb624d055f8>
In [43]: ax.set_xlabel('Stages')
```

Modifying the y-axis consists of the same process, substituting y for x in the above. The axes class has a set method that allows batch setting of plot properties. From the prior example, we could also have written:

```
props = {
'title': 'My first matplotlib plot',
'xlabel': 'Stages'
}
ax.set(**props)
```

Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the label argument when adding each piece of the plot:

```
In [44]: from numpy.random import randn
In [45]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>]
In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>]
In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>]
```

Once you've done this, you can either call ax.legend() or plt.legend() to automatically create a legend.

```
In [49]: ax.legend(loc='best')
```

The legend method has several other choices for the location loc argument. See the docstring (with ax.legend?) for more information.

The loc tells matplotlib where to place the plot. If you aren't picky, 'best' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or label='_nolegend_'.

Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes. You can add annotations and text using the text, arrow, and annotate functions. text draws text at given coordinates (x, y) on the plot with optional custom styling:

```
ax.text(x, y, 'Hello world!',
family='monospace', fontsize=10)
```

Annotations can draw both text and arrows arranged appropriately. As an example, let's plot the closing S&P 500 index price since 2007 (obtained from Yahoo! Finance) and annotate it with some of the important dates from the 2008–2009 financial crisis.

You can most easily reproduce this code example in a single cell in a Jupyter notebook.

```
from datetime import datetime
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True)
spx = data['SPX']
spx.plot(ax=ax, style='k-')
crisis_data = [
(datetime(2007, 10, 11), 'Peak of bull market'),
(datetime(2008, 3, 12), 'Bear Stearns Fails'),
(datetime(2008, 9, 15), 'Lehman Bankruptcy')
]
for date, label in crisis_data:
ax.annotate(label, xy=(date, spx.asof(date) + 75),
xytext=(date, spx.asof(date) + 225),
```

```
arrowprops=dict(facecolor='black', headwidth=4, width=2,
headlength=4),
horizontalalignment='left', verticalalignment='top')
# Zoom in on 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])
ax.set_title('Important dates in the 2008-2009 financial crisis')
```

There are a couple of important points to highlight in this plot: the ax.annotate method can draw labels at the indicated x and y coordinates. We use the set_xlim and set_ylim methods to manually set the start and end boundaries for the plot rather than using matplotlib's default. Lastly, ax.set_title adds a main title to the plot.

See the online matplotlib gallery for many more annotation examples to learn from.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as *patches*. Some of these, like Rectangle and Circle, are found in matplotlib.pyplot, but the full set is located in matplotlib.patches.

To add a shape to a plot, you create the patch object shp and add it to a subplot by calling ax.add_patch(shp):

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
color='g', alpha=0.5)
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

Saving Plots to File

You can save the active figure to file using plt.savefig. This method is equivalent to the figure object's savefig instance method. For example, to save an SVG version of a figure, you need only type:

```
plt.savefig('figpath.svg')
```

The file type is inferred from the file extension. So if you used .pdf instead, you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: dpi, which controls the dots-per-inch resolution, and bbox_inches, which can trim the whitespace around the actual figure. To get the same plot as a PNG with minimal whitespace around the plot and at 400 DPI, you would do:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

savefig doesn't have to write to disk; it can also write to any file-like object, such as a BytesIO:

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

See Table 14-13 for a list of some other options for savefig.

Table 14-13: Figure.savefig options

Argument	Description
Fname	String containing a file path or a Python file-like object. The figure format is inferred from the file extension (e.g., .pdf, .png).
Dpi	The figure resolution in dots per inch; defaults to 100 but can be configured.
facecolor, edgecolor	Colors of the figure background outside the subplots; defaults to 'w' (white).

Argument	Description
Format	Explicit file format to use (e.g., 'png', 'pdf', 'svg', 'ps', 'eps').
bbox_inches	Portion of the figure to save; use 'tight' to trim empty space around the figure.

Matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. One way to modify the configuration programmatically from Python is to use the rc method; for example, to set the global default figure size to be 10 × 10, you could enter:

```
plt.rc('figure', figsize=(10, 10))
```

The first argument to rc is the component you wish to customize, such as 'figure', 'axes', 'xtick', 'ytick', 'grid', 'legend', or many others.

After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

```
font_options = {'family' : 'monospace',
'weight' : 'bold',
'size' : 'small'}
plt.rc('font', **font_options)
```

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file *matplotlibrc* in the *matplotlib/mpl-data* directory. If you customize this file and place it in your home directory titled *.matplotlibrc*, it will be loaded each time you use matplotlib.

As we'll see in the next section, the seaborn package has several built-in plot themes or *styles* that use matplotlib's configuration system internally.

Plotting with pandas and seaborn

matplotlib can be a fairly low-level tool. You assemble a plot from its base components: the data display (i.e., the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations.

In pandas we may have multiple columns of data, along with row and column labels. pandas itself has built-in methods that simplify creating visualizations from DataFrame and Series objects. Another library is seaborn, a statistical graphics library created by Michael Waskom. Seaborn simplifies creating many common visualization types.

Importing seaborn modifies the default matplotlib color schemes and plot styles to improve readability and aesthetics. Even if you do not use the seaborn API, you may prefer to import seaborn as a simple way to improve the visual aesthetics of general matplotlib plots.

Line Plots

Series and DataFrame each have a plot attribute for making some basic plot types. By default, plot() makes line plots

```
In [60]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100,
10))
In [61]: s.plot()
```

The Series object's index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing use_index=False. The x-axis ticks and limits can be adjusted with the xticks and xlim options, and y-axis respectively with yticks and ylim. See Table 9-3 for a full listing of plot options. I'll

comment on a few more of them throughout this section and leave the rest to you to explore.

Most of pandas's plotting methods accept an optional ax parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout.

DataFrame's plot method plots each of its columns as a different line on the same subplot, creating a legend automatically

```
In [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
....: columns=['A', 'B', 'C', 'D'],
....: index=np.arange(0, 100, 10))
In [63]: df.plot()
```

The plot attribute contains a "family" of methods for different plot types. For exam-

ple, df.plot() is equivalent to df.plot.line(). We'll explore some of these methods

next.

Additional keyword arguments to plot are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

Table 14-14: Series.plot method arguments

Argument	**Description**
Label	Label for the plot legend
Ax	A matplotlib subplot object to plot on; if not provided, uses the active subplot
Style	Style string (e.g., 'ko--') passed to matplotlib
Alpha	Plot fill opacity (range: 0 to 1)
Kind	Type of plot: 'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie'
Logy	Use logarithmic scaling on the y-axis

Argument	Description
use_index	Use the object's index for tick labels
Rot	Rotation of tick labels (range: 0 to 360 degrees)
Xticks	Values to use for x-axis ticks
Yticks	Values to use for y-axis ticks
Xlim	Limits for the x-axis (e.g., [0, 10])
Ylim	Limits for the y-axis
Grid	Display axis grid (enabled by default)

DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots. See Table 9-4 for more on these.

Table 9-4: DataFrame-specific plot arguments

Argument	Description
Subplots	Plot each DataFrame column in a separate subplot
Sharex	If subplots=True, share the same x-axis (links ticks and limits)
Sharey	If subplots=True, share the same y-axis
Figsize	Size of the figure to create, specified as a tuple (e.g., (10, 6))
Title	Title of the plot, provided as a string
Legend	Add a subplot legend (True by default)
sort_columns	Plot columns in alphabetical order; by default, uses the existing column order

Bar Plots

The plot.bar() and plot.barh() make vertical and horizontal bar plots, respectively. In this case, the Series or DataFrame index will be used as the x (bar) or y (barh) ticks

```
In [64]: fig, axes = plt.subplots(2, 1)
In [65]: data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)
```

```
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb62493d470>
```

```
In [67]: data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

The options color='k' and alpha=0.7 set the color of the plots to black and use partial transparency on the filling.

With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value.

```
In [69]: df = pd.DataFrame(np.random.rand(6, 4),
....: index=['one', 'two', 'three', 'four', 'five', 'six'],
....: columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
In [70]: df
```

```
Out[70]:
Genus A B C D
one 0.370670 0.602792 0.229159 0.486744
two 0.420082 0.571653 0.049024 0.880592
three 0.814568 0.277160 0.880316 0.431326
four 0.374020 0.899420 0.460304 0.100843
five 0.433270 0.125107 0.494675 0.961825
six 0.601648 0.478576 0.205690 0.560547
```

```
In [71]: df.plot.bar()
```

Note that the name “Genus” on the DataFrame’s columns is used to title the legend.

We create stacked bar plots from a DataFrame by passing stacked=True, resulting in the value in each row being stacked together

```
In [73]: df.plot.barh(stacked=True, alpha=0.5)
```

A useful recipe for bar plots is to visualize a Series’s value frequency using value_counts: s.value_counts().plot.bar().

Returning to the tipping dataset used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day. I load the data using read_csv and make a cross-tabulation by day and party size:

```
In [75]: tips = pd.read_csv('examples/tips.csv')
In [76]: party_counts = pd.crosstab(tips['day'], tips['size'])
In [77]: party_counts
```

```
Out[77]:
size 1 2 3 4 5 6
day
Fri 1 16 1 1 0 0
Sat 2 53 18 13 1 0
Sun 0 39 15 18 3 1
Thur 1 48 4 5 1 3
```

```
# Not many 1- and 6-person parties
In [78]: party_counts = party_counts.loc[:, 2:5]
```

Then, normalize so that each row sums to 1 and make the plot:

```
# Normalize to sum to 1
In [79]: party_pcts = party_counts.div(party_counts.sum(1), axis=0)
In [80]: party_pcts
```

```
Out[80]:
size 2 3 4 5
day
Fri 0.888889 0.055556 0.055556 0.000000
Sat 0.623529 0.211765 0.152941 0.011765
Sun 0.520000 0.200000 0.240000 0.040000
Thur 0.827586 0.068966 0.086207 0.017241
```

```
In [81]: party_pcts.plot.bar()
```

So you can see that party sizes appear to increase on the weekend in this dataset.

With data that requires aggregation or summarization before making a plot, using the seaborn package can make things much simpler. Let's look now at the tipping percentage by day with seaborn.

```
In [83]: import seaborn as sns
In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])
In [85]: tips.head()
```

Out[85]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.063204
1	10.34	1.66	No	Sun	Dinner	3	0.191244
2	21.01	3.50	No	Sun	Dinner	3	0.199886
3	23.68	3.31	No	Sun	Dinner	2	0.162494
4	24.59	3.61	No	Sun	Dinner	4	0.172069

```
In [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

Plotting functions in seaborn take a data argument, which can be a pandas DataFrame. The other arguments refer to column names. Because there are multiple observations for each value in the day, the bars are the average value of tip_pct. The black lines drawn on the bars represent the 95% confidence interval (this can be configured through optional arguments).

seaborn.barplot has a hue option that enables us to split by an additional categorical value

```
In [88]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors. You can switch between different plot appearances using seaborn.set:

```
In [90]: sns.set(style="whitegrid")
```

Advanced Histograms and Density Plots

A histogram is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Using the tipping data from before, we can make a histogram of tip percentages of the total bill using the plot.hist method on the Series

```
In [92]: tips['tip_pct'].plot.hist(bins=50)
```

A related plot type is a *density plot*, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. The usual procedure is to approximate this distribution as a mixture of "kernels",that is, simpler distributions like the normal distribution. Thus, density plots are also known as kernel density estimate (KDE) plots. Using plot.kde makes a density plot using the conventional mixture-of-normals estimate

```
In [94]: tips['tip_pct'].plot.density()
```

Seaborn makes histograms and density plots even easier through its distplot method, which can plot both a histogram and a continuous density estimate simultaneously. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions:

```
In [96]: comp1 = np.random.normal(0, 1, size=200)
In [97]: comp2 = np.random.normal(10, 2, size=200)
In [98]: values = pd.Series(np.concatenate([comp1, comp2]))
In [99]: sns.distplot(values, bins=100, color='k')
```

Advanced Scatter or Point Plots

Point plots or scatter plots can be a useful way of examining the relationship between two one-dimensional data series. For example, here we load the macrodata dataset from the statsmodels project, select a few variables, then compute log differences:

```
In [100]: macro = pd.read_csv('examples/macrodata.csv')
In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]
In [102]: trans_data = np.log(data).diff().dropna()
In [103]: trans_data[-5:]
```

```
Out[103]:
cpi m1 tbilrate unemp
198 -0.007904 0.045361 -0.396881 0.105361
199 -0.021979 0.066753 -2.277267 0.139762
```

```
200 0.002340 0.010286 0.606136 0.160343
201 0.008419 0.037461 -0.200671 0.127339
202 0.008894 0.012202 -0.405465 0.042560
```

We can then use seaborn's regplot method, which makes a scatter plot and fits a linear regression line:

```
In [105]: sns.regplot('m1', 'unemp', data=trans_data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb613720be0>
In [106]: plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))
```

In exploratory data analysis it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a *pairs* plot or *scatter plot matrix*. Making such a plot from scratch is a bit of work, so seaborn has a convenient pairplot function, which supports placing histograms or density estimates of each variable along the diagonal

```
In [107]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```

You may notice the plot_kws argument. This enables us to pass down configuration options to the individual plotting calls on the off-diagonal elements. Check out the seaborn.pairplot docstring for more granular configuration options.

Facet Grids and Categorical Data

What about datasets where we have additional grouping dimensions? One way to visualize data with many categorical variables is to use a *facet grid*. Seaborn has a useful built-in function factorplot that simplifies making many kinds of faceted plots

```
In [108]: sns.factorplot(x='day', y='tip_pct', hue='time', col='smoker',
.....: kind='bar', data=tips[tips.tip_pct < 1])
```

Instead of grouping by 'time' by different bar colors within a facet, we can also expand the facet grid by adding one row per time value

```
In [109]: sns.factorplot(x='day', y='tip_pct', row='time',
.....: col='smoker',
```

```
.....: kind='bar', data=tips[tips.tip_pct < 1])
```

factorplot supports other plot types that may be useful depending on what you are trying to display. For example, box plots (which show the median, quartiles, and outliers) can be an effective visualization type:

```
In [110]: sns.factorplot(x='tip_pct', y='day', kind='box',
.....: data=tips[tips.tip_pct < 0.5])
```

You can create your own facet grid plots using the more general seaborn.FacetGrid class. See the seaborn documentation for more.

As is common with open source, there are a plethora of options for creating graphics in Python (too many to list). Since 2010, much development effort has been focused on creating interactive graphics for publication on the web. With tools like Bokeh and Plotly, it's now possible to specify dynamic, interactive graphics in Python that are destined for a web browser.

For creating static graphics for print or web, I recommend defaulting to matplotlib and add-on libraries like pandas and seaborn for your needs. For other data visualization requirements, it may be useful to learn one of the other available tools out there.

I encourage you to explore the ecosystem as it continues to involve and innovate into the future.

QUESTIONS

1. What is the purpose of data visualization in data science?
2. Name two advantages of using Seaborn over Matplotlib for statistical plotting.
3. How would you display the first five rows of a DataFrame named `df` before plotting?
4. Write a Python command to create a histogram of a column named `age` using Pandas.
5. What type of plot would best show the distribution of a single numerical variable?
6. Which Matplotlib function is used to display a plot?
7. What does the `plt.subplot()` function do in Matplotlib?
8. Write the code to create a basic line plot using Matplotlib for lists `x` and `y`.
9. How do you change the color and line style in a Matplotlib plot?
10. Explain what a boxplot shows in a dataset.
11. What is the difference between a bar chart and a histogram?
12. Give an example of an interactive visualization library in Python.
13. What argument would you use in Seaborn's `sns.scatterplot()` to change the marker color?
14. How do you create a correlation heatmap using Seaborn?
15. What is the role of `figsize` in plotting with Matplotlib?
16. How would you save a plot as an image file using Matplotlib?
17. What does `plotly.express` simplify compared to the standard `plotly.graph_objects`?

18. Why is it important to style and customize plots in data science storytelling?

MODULE 15

TIME SERIES: AN OVERVIEW

In the world of data, *time series* is where timestamps and data shake hands. Whether you're tracking stock prices, recording temperatures, monitoring machine sensors, or following your fitness steps, you're likely dealing with time series data.

Time series analysis is a critical aspect of data science, particularly useful in forecasting, trend analysis, and anomaly detection across domains like finance, weather prediction, healthcare, and industrial monitoring.

By the end of this module, you should be able to:

- Understand what time series data is and identify its various forms and frequencies.
- Use Python's datetime, time, and calendar modules for date and time manipulation.
- Parse, format, and convert date/time strings using both standard Python and dateutil.
- Work effectively with Pandas' time series tools including DatetimeIndex, Timestamp, and to datetime().
- Handle missing values and duplicate timestamps in time series.
- Generate custom date ranges and shift or resample time series to different frequencies.
- Perform indexing, slicing, arithmetic, and aggregation on time series data.

What is time series data?

Time series data is an essential form of structured data used across various disciplines, including finance, economics, ecology,

neuroscience, and physics. Any data observed or measured at multiple time points constitutes a time series.

Time series data is simply a sequence of observations recorded at regular or irregular intervals over time. Think of:

- Stock prices (every second, minute, or day) 📈
- Temperature readings (hourly/daily) 🌡️
- Website traffic (per minute/hour)

These data can be categorized into:

1. **Timestamped Data** – Events logged at specific moments (e.g., 2024-05-01 14:30:00).
   Example: Server logs, transaction timestamps.
2. **Fixed Periods** – Aggregated data over intervals (e.g., monthly sales).
   Example: Quarterly GDP reports, daily active users.
3. **Time Intervals** – Duration between events (e.g., session length).
   Example: Time between customer purchases.
4. **Elapsed or experimental time**: Measures time relative to a starting point (e.g., tracking the expansion of a cookie in the oven every second).

The most commonly used type is timestamp-based time series data. Pandas provides tools for handling both fixed-frequency and irregular time series, allowing for efficient slicing, aggregation, and resampling.

Working with Dates and Times in Python

The Python standard library provides essential modules for date and time handling:

- **datetime**: Handles date and time objects.
- **time**: Works with time-related functions.
- **calendar**: Offers calendar-related functionalities.

Using datetime module

The datetime module is Python's core tool for handling dates and times. Key classes:

- datetime.datetime – Combines date and time (e.g., 2024-05-01 14:30:00).
- datetime.timedelta – Represents time differences (e.g., "7 days").

Why it matters:

- Calculate future/past dates (e.g., "What's the date 30 days from now?").
- Measure time intervals (e.g., "How long did this process take?")

```
from datetime import datetime
now = datetime.now()
print(now)  # Output: 2025-02-27 10:30:45.123456
```

Using time Module

The time module deals with low-level time operations:

- time.time() – Returns Unix timestamp (seconds since Jan 1, 1970).
- time.sleep() – Pauses program execution.

```
import time

# Get current Unix timestamp
timestamp = time.time()
print("Current timestamp:", timestamp)

# Pause execution for 3 seconds
print("Sleeping for 3 seconds...")
time.sleep(3)
print("Done sleeping!")
```

Why it matters:

- Benchmarking code performance.
- Scheduling tasks at precise intervals.

Using calendar Module

The calendar module handles date-related calculations:

- calendar.isleap(year) – Checks for leap years.
- calendar.weekday(year, month, day) – Returns day of the week (0=Monday).

```
import calendar
# Check if a year is a leap year
print(calendar.isleap(2024))  # True
print(calendar.isleap(2023))  # False
# Get weekday (0=Monday, 6=Sunday)
print(calendar.weekday(2024, 7, 15))  # 0 (Monday)
```

Why it matters:

- Validating dates (e.g., "Is February 29, 2023 valid?").
- Planning weekly/monthly reporting.

Using the timedelta Class

The timedelta class in Python's datetime module enables date and time arithmetic, making it easy to compute differences between dates or shift them forward/backward. For example, you can:

- Add or subtract days, seconds, or microseconds from a date.
- Calculate durations (e.g., "How many days until the project deadline?").
- Generate sequences of dates (e.g., "Every 7 days for the next month").

A timedelta object represents a duration, not an absolute time, and supports operations like:

Date Arithmetic

```
future_date = datetime(2024, 1, 1) + timedelta(days=7)  # Adds 1 week
past_date = datetime.now() - timedelta(hours=3)        # Subtracts 3 hours
```

Duration Comparisons

```
delta1 = timedelta(days=1)
delta2 = timedelta(hours=36)
print(delta2 > delta1)  # True (36 hours > 24 hours)
```

Time Interval Calculations

```
start = datetime(2024, 1, 1)
end = datetime(2024, 1, 15)
project_duration = end - start  # Returns timedelta(days=14)
```

Component Extraction

```
delta = timedelta(days=2, hours=5)
print(delta.days)       # 2 (total full days)
print(delta.seconds)    # 18000 (5 hours in seconds)
```

Scaling Operations

```
double_delta = timedelta(days=1) * 2   # 2 days
half_hour = timedelta(hours=1) / 2     # 30 minutes
```

Table 15-1: Key datetime Module Types

Type	Description
Date	Stores calendar date (year, month, day).
Time	Stores time of day (hours, minutes, seconds).
Datetime	Combines date and time.
timedelta	Represents the difference between dates/times.
Tzinfo	Handles time zone information.

Converting Between Strings and Datetime Objects

Working with dates often requires switching between human-readable strings (like "2024-07-15") and programmable datetime objects. Python makes this easy with two key methods:

1. strftime() (Datetime → String)

Formats a datetime object into a custom string representation.

Uses format codes like %Y (year), %m (month), %d (day), etc.
Example:

```
from datetime import datetime
now = datetime.now()
formatted_date = now.strftime("%Y-%m-%d")  # "2024-07-15"
formatted_time = now.strftime("%H:%M:%S")  # "14:30:00"
```

Use Case:

- Generating report filenames ("sales_2024-07-15.csv")
- Displaying dates in UIs/dashboards

2. strptime() (String → Datetime)

Parses a string into a datetime object. Requires specifying the exact format of the input string
Example:

```
date_str = "July 15, 2024"
parsed_date = datetime.strptime(date_str, "%B %d, %Y")
```

Use Case:

- Cleaning messy date data from CSVs/APIs
- Standardizing dates before analysis

Use dateutil.parser for fuzzy parsing of inconsistent formats!

Why This Matters:

- 80% of real-world date data starts as strings
- Proper conversion enables sorting, filtering and time-based calculations

Handling Time Series in Pandas

Pandas simplifies working with large time series datasets. The to_datetime() function efficiently converts date strings into Pandas DatetimeIndex:

```
import pandas as pd
```

```
date_strings = ['2011-07-06 12:00:00', '2011-08-06 00:00:00', None]
datetime_index = pd.to_datetime(date_strings)
print(datetime_index)
```

Missing Values in Time Series

Pandas handles missing values (None, empty strings, etc.) and represents them as NaT (Not a Time):

```
print(datetime_index[2]) # Output: NaT
print(pd.isnull(datetime_index)) # Output: array([False, False, True], dtype=bool)
```

Locale-Specific Date Formatting

Pandas supports locale-specific formatting using various directives:

Table 15-2: Common strftime Format Codes for Date/Time Formatting

Type	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Full date and time (e.g., 'Tue 01 May 2012 04:20:57 PM')
%p	Locale equivalent of AM or PM
%x	Locale-appropriate formatted date (e.g., '05/01/2012' in the U.S.)
%X	Locale-appropriate time (e.g., '04:24:12 PM')

Time Series Basics in Pandas

A basic time series in Pandas is a Series indexed by timestamps:

```
from datetime import datetime
import numpy as np
# Creating a time series
dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
        datetime(2011, 1, 7), datetime(2011, 1, 8),
        datetime(2011, 1, 10), datetime(2011, 1, 12)]
```

```
ts = pd.Series(np.random.randn(6), index=dates)
print(ts)
```

Indexing and Subsetting

Pandas allows easy selection and slicing of time series data:

```
print(ts['2011-01-10'])  # Selects a specific date
print(ts['2011'])  # Selects all data for the year 2011
print(ts['2011-05'])  # Selects all data for May 2011
```

Using datetime objects for slicing:

```
print(ts[datetime(2011, 1, 7):])
```

Performing arithmetic operations between time series aligns them on timestamps:

```
print(ts + ts[::2])
```

Pandas Timestamp and DatetimeIndex

Timestamps in Pandas use NumPy's datetime64 type at nanosecond resolution:

```
print(ts.index.dtype)  # Output: dtype('<M8[ns]')
```

Individual values in a DatetimeIndex are Timestamp objects:

```
stamp = ts.index[0]
print(stamp)  # Output: Timestamp('2011-01-02 00:00:00')
```

By leveraging Python's built-in and Pandas functionalities, handling and analyzing time series data becomes seamless across various applications.

Time Series with Duplicate Indices

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/200
0', '1/3/2000'])
In [64]: dup_ts = pd.Series(np.arange(5), index=dates)
In [65]: dup_ts
```

```
Out[65]:
2000-01-01  0
2000-01-02  1
2000-01-02  2
```

```
2000-01-02    3
2000-01-03    4
dtype: int64
```

We can tell that the index is not unique by checking its is_unique property:

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:

```
In [67]: dup_ts['1/3/2000']  # not duplicated
Out[67]: 4
In [68]: dup_ts['1/2/2000']  # duplicated
Out[68]:
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

Suppose you wanted to aggregate the data having non-unique timestamps. One way to do this is to use groupby and pass level=0:

```
In [69]: grouped = dup_ts.groupby(level=0)
In [70]: grouped.mean()
Out[70]:
2000-01-01    0
2000-01-02    2
2000-01-03    4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

Date Ranges, Frequencies, and Shifting

Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications, this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately, pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed-frequency date ranges. For example, you can convert the sample time series to be fixed daily frequency by calling resample:

```
In [72]: ts
Out[72]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

The string 'D' is interpreted as daily frequency.

Generating Date Ranges

While I used it previously without explanation, pandas.date_range is responsible for generating a DatetimeIndex with an indicated length according to a particular frequency:

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')
In [75]: index
```

```
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
```

```
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

By default, date_range generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
```

```
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')
```

```
In [77]: pd.date_range(end='2012-06-01', periods=20)
```

```
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the 'BM' frequency (business end of month; see more complete listing of frequencies in Table 14-3) and only dates falling on or inside the date interval will be included:

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
```

```
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

Table 14-3: Base time series frequencies (not comprehensive)

Alias	Offset type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
S	Second	Secondly
L or ms	Milli	Millisecond (1/1,000 of 1 second)
U	Micro	Microsecond (1/1,000,000 of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
W-MON, W-TUE,	Week	Weekly on given day of week (MON, TUE, WED, THU,

Alias	Offset type	Description
...		FRI, SAT, or SUN)
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month (e.g., WOM-3FRI for the third Friday of each month)
Q-JAN, Q-FEB, ...	QuarterEnd	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
QS-JAN, QS-FEB, ...	QuarterBegin	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
A-JAN, A-FEB, ...	YearEnd	Annual dates anchored on last calendar day of given month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BA-JAN,	BusinessYearEnd	Annual dates anchored on last

Alias	Offset type	Description
BA-FEB, ...		weekday of given month
AS-JAN, AS-FEB, ...	YearBegin	Annual dates anchored on first day of given month
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Annual dates anchored on first weekday of given month

date_range by default preserves the time (if any) of the start or end timestamp:

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods=5)
```

```
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

Sometimes you will have start or end dates with time information but want to generate a set of timestamps normalized to midnight as a convention. To do this, there is a normalize option:

```
In [80]: pd.date_range('2012-05-02 12:56:31', periods=5, normalize=True)
```

```
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
               '2012-05-06'],
              dtype='datetime64[ns]', freq='D')
```

Frequencies and Date Offsets

Frequencies in pandas are composed of a base frequency and a multiplier. Base frequencies are typically referred to by a string alias, like 'M' for monthly or 'H' for hourly. For each base frequency, there is an object defined generally referred to as a date

offset. For example, hourly frequency can be represented with the Hour class:

```
In [81]: from pandas.tseries.offsets import Hour, Minute
In [82]: hour = Hour()
In [83]: hour
```

```
Out[83]: <Hour>
```

You can define a multiple of an offset by passing an integer:

```
In [84]: four_hours = Hour(4)
In [85]: four_hours
```

```
Out[85]: <4 * Hours>
```

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like 'H' or '4H'. Putting an integer before the base frequency creates a multiple:

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')
```

```
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

Many offsets can be combined together by addition:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

Similarly, you can pass frequency strings, like '1h30min', that will effectively be parsed to the same expression:

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
```

```
 '2000-01-01 09:00:00', '2000-01-01 10:30:00',
 '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
dtype='datetime64[ns]', freq='90T')
```

Some frequencies describe points in time that are not evenly spaced. For example, 'M' (calendar month end) and 'BM' (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. We refer to these as anchored offsets.

Week of Month Dates

One useful frequency class is "week of month," starting with WOM. This enables you to get dates like the third Friday of each month:

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')
In [90]: list(rng)
```

```
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-07-20 00:
```

QUESTIONS

1. What function is used in pandas to convert a column to datetime format?
2. How do you set a datetime column as the index of a DataFrame?
3. What method is used to fill in missing dates using the previous available value?
4. How would you select all rows from January 2025 in a time-indexed DataFrame?
5. Which attributes would you use to extract the day of the week and the month from a datetime index?

MODULE 16

ADVANCED TIME SERIES ANALYSIS WITH PYTHON

This module provides a comprehensive introduction to time series analysis, covering fundamental concepts, preprocessing techniques, model building, and evaluation. By mastering these techniques, you will be well-equipped to tackle real-world time series problems using Python. When readers complete the module, they will need to be able to:

1. **Understand Time Series Fundamentals:**
    - Describe time series and how it is used in finance, economics, healthcare, retail and meteorology.
    - Point out the main features in time series data: trend, seasonal patterns, cyclic trends and anything remaining(residuals).
2. **Preprocess Time Series Data:**
    - Manage missing values in data using interpolation methods, filling forward or backward or removing the values.
    - Increase or decrease the frequency of your time series data by resampling..
    - Remove noise by applying methods like moving average and exponential smoothing to your dataset.
    - Use Min-Max Scaling or Standardization to scale time series data before you start modeling.
3. **Perform Time Series Decomposition:**
    - Understand how to use additive and multiplicative approaches to separate a time series into its trend, seasonal and remaining parts.

- Decompose time series using the seasonal_decompose function from the modifications statsmodels.

4. **Test for Stationarity and Make Data Stationary:**
    - Familiarize yourself with the term stationarity and why it supports time series modeling.
    - Check whether the series is stationary by applying the Augmented Dickey-Fuller (ADF) test.
    - Make a time series stationary by using differencing and transformations.
5. **Analyze Autocorrelation and Partial Autocorrelation:**
    - Study ACF and PACF to determine the right parameters for an ARIMA model.
    - Use these two functions plot_acf and plot_pacf provided by Python to create and analyze the relationships.
6. **Build and Evaluate ARIMA Models:**
    - Be familiar with the concepts of ARIMA and practice choosing (p, d, q).
    - Use the ARIMA function of the statsmodels library allows you to perform ARIMA analyses in Python.
    - Interpret model results and forecast future values using ARIMA.
7. **Extend ARIMA with SARIMA Models:**
    - Understand SARIMA (Seasonal ARIMA), a model designed to handle seasonality.
    - Apply SARIMA to time series data, adjusting for seasonal patterns and trends.
    - Implement and forecast with SARIMA using the SARIMAX class in Python.

8. **Use Exponential Smoothing for Time Series Forecasting:**
    - Learn about different types of exponential smoothing, including simple, Holt's linear trend, and Holt-Winters method.
    - Implement Holt-Winters Exponential Smoothing for data with trend and seasonality.
9. **Model Multiple Time Series Using VAR:**
    - Grasp the concept of Vector Autoregression (VAR) and its use in modeling multiple interdependent time series variables.
    - Implement VAR in Python for forecasting and policy analysis with multiple related variables.
10. **Forecast Volatility with GARCH Models:**
    - Understand and apply GARCH (Generalized Autoregressive Conditional Heteroskedasticity) models to analyze volatility and forecast future volatility in financial time series.
    - Utilize GARCH models for risk management and portfolio optimization.

INTRODUCTION

Definition of Time Series

A time series consists of a sequence of observations recorded at specific time intervals, such as daily stock prices, monthly sales figures, or annual rainfall data. The goal of time series analysis is to extract meaningful insights and patterns from this data.

Practical Applications

Time series analysis is widely used across various fields:

- **Finance:** Stock market predictions, risk assessment.
- **Economics:** GDP forecasting, employment trends.

- **Healthcare:** Patient health monitoring, outbreak detection.
- **Retail:** Demand forecasting, inventory control.
- **Meteorology:** Weather and climate prediction.

Key Components of Time Series

A time series can be broken down into distinct components:

Trend

The long-term directional movement in the data, which can be increasing, decreasing, or stable.

Seasonality

Recurring patterns that occur at regular intervals, such as monthly, quarterly, or yearly trends.

Cyclic Patterns

Variations that occur over irregular periods, often influenced by external factors like economic cycles.

Residual (Irregular Component)

Unpredictable fluctuations in the data that cannot be explained by trend, seasonality, or cyclic behavior.

PREPROCESSING TIME SERIES DATA

Handling Missing Values

- **Interpolation:** Filling gaps using methods like linear or spline interpolation.
- **Forward/Backward Fill:** Using neighboring values to replace missing data.
- **Removing Missing Data:** Deleting rows with missing values if necessary.

Resampling

- **Upsampling:** Increasing data frequency (e.g., converting monthly data to daily).

- **Downsampling:** Reducing data frequency (e.g., daily data aggregated to monthly).

Smoothing Techniques

- **Moving Average:** Reducing short-term fluctuations to highlight trends.
- **Exponential Smoothing:** Assigning higher weights to recent data points.

Data Scaling

- **Min-Max Scaling:** Rescaling values within a fixed range (e.g., 0 to 1).
- **Standardization:** Adjusting data to have zero mean and unit variance.

TIME SERIES DECOMPOSITION

Time series decomposition breaks down a time series into its key components: **Trend**, **Seasonality**, and **Residual (Noise)**. The two main approaches are **additive** and **multiplicative** decomposition.

Additive Decomposition

Represents the time series as the sum of its components: This is applied when the magnitude of seasonality does not change with the trend and when the trend and seasonal effects are independent of each other.

$$Y(t) = \text{Trend}(t) + \text{Seasonality}(t) + \text{Residual}(t)$$

Example:

If a time series has:

- Trend: 100 units
- Seasonality: +20 units (peak season)
- Residual: +5 units (random fluctuation)

Then:

$$Y(t) = 100 + 20 + 5 = 125$$

Multiplicative Decomposition

Expresses the series as the product of its components. **this is applied** when the **seasonal effect scales with the trend** (e.g., higher trend → larger seasonal swings). It is common in economic and financial data (e.g., sales growing over time with amplified seasonal peaks).

$$Y(t) = \text{Trend}(t) \times \text{Seasonality}(t) \times \text{Residual}(t)$$

Example:

If a time series has:

- Trend: 100 units
- Seasonality: 1.2x (20% increase in peak season)
- Residual: 1.05x (5% random fluctuation)

Then:

$$Y(t) = 100 \times 1.2 \times 1.05 = 126$$

Implementing Decomposition in Python

```
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(series, model='additive', period=12)
result.plot()
```

STATIONARITY IN TIME SERIES

Definition of Stationarity

A time series is stationary if its statistical properties (mean, variance, autocorrelation) remain constant over time.

Why Stationarity Matters

Most time series forecasting models assume stationarity; non-stationary data may yield inaccurate predictions.

Testing for Stationarity

- **Augmented Dickey-Fuller (ADF) Test:** Determines whether a time series is stationary.

```
from statsmodels.tsa.stattools import adfuller
result = adfuller(series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
```

Making Data Stationary

- **Differencing:** Subtracting consecutive observations.
- **Transformation:** Applying log or square root transformations to stabilize variance.

AUTOCORRELATION AND PARTIAL AUTOCORRELATION

Autocorrelation Function (ACF)

Measures the correlation between a time series and its past values.

Partial Autocorrelation Function (PACF)

Evaluates the direct correlation between a time series and its lagged values, removing intermediate effects.

Plotting ACF and PACF

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

plot_acf(series, lags=40)
plot_pacf(series, lags=40)
```

TIME SERIES MODELS

Time series modeling involves developing mathematical models to forecast future values based on previously observed values.

One of the most widely used models in this space is ARIMA, which stands for AutoRegressive Integrated Moving Average. This powerful model combines three key ideas: Autoregression (AR), Integration (I), and Moving Average (MA).

Let's break this down:

Autoregression (AR) uses the dependency between an observation and a number of lagged (previous) observations. For example, today's sales might depend on sales from the past three days.

Integration (I) refers to the differencing of raw observations (subtracting an observation from its previous one) to make the time series stationary,essentially, to remove trends and make the data stable over time.

Moving Average (MA) incorporates the dependency between an observation and a residual error from a moving average model applied to lagged observations.

ARIMA models are denoted as ARIMA(p, d, q),

where:

p is the number of lag observations (AR terms),

d is the degree of differencing (how many times data is differenced to achieve stationarity),

q is the size of the moving average window (MA terms).

Example: Imagine you're analyzing daily electricity consumption. If you notice a trend and some random fluctuations, an ARIMA model might help. Suppose ARIMA(1,1,1) is selected,this means you take one difference of the data (to remove trend), use one lag of the past data, and model one lag of the forecast errors. With this, you can forecast tomorrow's energy usage using a formula that blends past observations and errors in a mathematically sound way.

Before using ARIMA, it's important to ensure the time series is stationary. If not, differencing is applied. Tools like Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots help identify appropriate values for p and

q. Python's statsmodels library offers a simple implementation of ARIMA through ARIMA() and SARIMAX() classes.

Implementing ARIMA Timeseries forecasting in Python

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from pandas.plotting import register_matplotlib_converters

register_matplotlib_converters()

# Load a sample dataset (AirPassengers)
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv'
df = pd.read_csv(url, parse_dates=['Month'], index_col='Month')

# Visualize the data
plt.figure(figsize=(10, 4))
plt.plot(df, label='Monthly Passengers')
plt.title('Monthly Airline Passenger Counts (1949–1960)')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.legend()
plt.grid(True)
plt.show()
```

```
# Check if the series is stationary (a quick visual check)
# Differencing to make it stationary (d=1)
df_diff = df.diff().dropna()

# Plot ACF and PACF to identify AR (p) and MA (q)
plot_acf(df_diff)
plot_pacf(df_diff)
plt.show()
```

```
# Fit ARIMA model: let's say ARIMA(2,1,2)
model = ARIMA(df, order=(2, 1, 2))
model_fit = model.fit()

# Summary of the model
print(model_fit.summary())

# Forecast the next 12 months
forecast = model_fit.forecast(steps=12)
print("\nForecasted values:")
print(forecast)

# Plot forecast against actual
plt.figure(figsize=(10, 4))
plt.plot(df, label='Historical Data')
plt.plot(forecast.index, forecast, label='Forecast', color='red')
plt.title('ARIMA Forecast of Airline Passengers')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.legend()
plt.grid(True)
plt.show()
```

SARIMA Model

SARIMA (Seasonal AutoRegressive Integrated Moving Average) is an extension of ARIMA that supports seasonality in your time series. While ARIMA captures trends and patterns, SARIMA adds the ability to model seasonal effects that repeat over a fixed period (like months or quarters).

SARIMA is often denoted as: SARIMA(p, d, q)(P, D, Q, s)

p, d, q: non-seasonal ARIMA parameters.

P, D, Q: seasonal components.

s: the length of the season (e.g., 12 for monthly data with yearly seasonality).

Example Let's build a SARIMA model on the famous AirPassengers dataset (monthly data from 1949–1960).

```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.seasonal import seasonal_decompose

# Load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv'
df = pd.read_csv(url, parse_dates=['Month'], index_col='Month')

# Plot the data
df.plot(figsize=(10, 4), title='Monthly Airline Passengers (1949–1960)')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.grid(True)
plt.show()

# Decompose to visualize trend and seasonality
result = seasonal_decompose(df, model='multiplicative')
result.plot()
plt.tight_layout()
plt.show()

# Fit SARIMA model
# Choosing SARIMA(1,1,1)(1,1,1,12) based on seasonality of 12 months
model = SARIMAX(df,
                order=(1, 1, 1),
                seasonal_order=(1, 1, 1, 12),
                enforce_stationarity=False,
                enforce_invertibility=False)
sarima_fit = model.fit()

# Print the model summary
print(sarima_fit.summary())
```

```
# Forecast the next 12 months
forecast = sarima_fit.get_forecast(steps=12)
conf_int = forecast.conf_int()

# Plotting the forecast
plt.figure(figsize=(10, 4))
plt.plot(df, label='Observed')
plt.plot(forecast.predicted_mean, label='Forecast', color='red')
plt.fill_between(conf_int.index,
                 conf_int.iloc[:, 0],
                 conf_int.iloc[:, 1],
                 color='pink', alpha=0.3)
plt.title('SARIMA Forecast of Airline Passengers')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.legend()
plt.grid(True)
plt.show()
```

Exponential Smoothing

Exponential Smoothing is like putting sunglasses on noisy time series data,it helps you see the real trend without being blinded by the noise. It gives more weight to recent observations but doesn't completely ignore older ones.

There are three main types:

- Simple Exponential Smoothing (SES) – best for data with no trend or seasonality.
- Holt's Linear Trend Method – for data with a trend but no seasonality.
- Holt-Winters Method – for data with trend and seasonality. That's our hero today!

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
```

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv'
df = pd.read_csv(url, parse_dates=['Month'], index_col='Month')

# Plot the original data
df.plot(figsize=(10, 4), title='Monthly Airline Passengers')
plt.xlabel('Date')
plt.ylabel('Number of Passengers')
plt.grid(True)
plt.show()

# Apply Holt-Winters Exponential Smoothing
# We assume trend and seasonality are multiplicative (because the seasonal effect increases over time)
hw_model = ExponentialSmoothing(df['Passengers'],
                                trend='multiplicative',
                                seasonal='multiplicative',
                                seasonal_periods=12).fit()

# Forecast the next 12 months
forecast = hw_model.forecast(steps=12)

# Plot the forecast
plt.figure(figsize=(10, 4))
plt.plot(df['Passengers'], label='Observed')
plt.plot(forecast, label='Holt-Winters Forecast', color='red')
plt.title('Holt-Winters Exponential Smoothing Forecast')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.legend()
plt.grid(True)
plt.show()
```

Vector Autoregression (VAR)

Vector Autoregression (VAR) is a powerful statistical tool used for analyzing the dynamic relationships between multiple time series variables. Unlike a univariate model that relies only on its own history to expect a single outcome, VAR treats each variable as related to its history and to the histories of every other variable in the model. Thus, VAR plays an important role in economics and finance, since GDP, inflation and interest rates typically affect each other. When using a VAR model, one can forecast the effects of an interest rate change on inflation and the country's GDP growth, considering the links between them.

This means that without needing to know what is exogenous or endogenous in advance, VAR remains free from any economic theories. Nevertheless, as the number of variables rises in a VAR model, it can grow complicated and demanding and they are only suited to modeling linear interactions. Nevertheless, VAR has found wide application in forecasting, analyzing policies and examining relationships between variables (such as Granger causality). Libraries such as statsmodels provide an easy way in Python to use VAR, fit models, find the best lag length and predict values for several variables at one time. VAR is an effective tool for breaking down how various time series change and adapt as they interact over periods of time.

- Top of Form
- Bottom of Form

Use Case: Forecasting GDP and inflation jointly for policy planning.

Modeling GDP and Unemployment with VAR

Let's use a built-in macroeconomic dataset from statsmodels.

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
from statsmodels.tsa.api import VAR
from statsmodels.datasets import macrodata

# Load macroeconomic data
data = macrodata.load_pandas().data

# Select GDP and unemployment rate for example
df = data[['realgdp', 'unemp']]
df.index = pd.date_range(start='1959Q1', periods=len(df), freq='Q')

# Plot the variables
df.plot(title='Real GDP and Unemployment Rate', figsize=(10, 4))
plt.grid(True)
plt.show()

# Step 1: Difference the data to make it stationary
df_diff = df.diff().dropna()

# Step 2: Fit the VAR model
model = VAR(df_diff)
results = model.fit(maxlags=4, ic='aic')  # Let AIC choose best lag

# Step 3: Forecast 8 steps ahead
forecast = results.forecast(df_diff.values[-results.k_ar:], steps=8)

# Convert forecast to DataFrame
forecast_df = pd.DataFrame(forecast,
index=pd.date_range(start=df_diff.index[-1]+1, periods=8, freq='Q'),
                           columns=['realgdp', 'unemp'])

# Plot the forecast
df_diff.plot(title='Differenced Series', figsize=(10, 4))
forecast_df.plot(title='VAR Forecast (Differenced)', figsize=(10, 4), style='--')
plt.grid(True)
plt.show()
```

GARCH Model

The GARCH Model (Generalized Autoregressive Conditional Heteroskedasticity) is a powerful statistical tool designed to analyze and forecast volatility in time series data, particularly in financial markets. Unlike traditional models that assume constant variance, GARCH captures the phenomenon of volatility clustering - where periods of high volatility tend to persist, followed by periods of low volatility. This makes it exceptionally useful for risk management, derivative pricing, and portfolio optimization in finance.

The core GARCH(1,1) model represents conditional variance through three key components: long-term average volatility (ω), reaction to recent market shocks (α), and persistence of past volatility (β). While extremely valuable for volatility forecasting, GARCH has limitations including its inability to capture asymmetric effects (where negative shocks impact volatility differently than positive ones) and computational complexity with high-frequency data. More advanced variants like EGARCH and GJR-GARCH address some of these limitations by modeling asymmetric volatility responses.

```
from arch import arch_model
import pandas as pd

# Load financial returns data
returns = pd.read_csv("stock_returns.csv", index_col=0, parse_dates=True)

# Fit GARCH(1,1) model
model = arch_model(returns, vol="GARCH", p=1, q=1)
results = model.fit(update_freq=5)

# Display model summary
print(results.summary())

# Generate 5-day volatility forecast
```

```
forecast = results.forecast(horizon=5)
print(forecast.variance.iloc[-1])
```

This Python implementation demonstrates a typical GARCH workflow: loading financial returns data, fitting the GARCH(1,1) model, and generating volatility forecasts. The ARCH package provides a comprehensive toolkit for GARCH modeling and its extensions, making it invaluable for financial analysts and quantitative researchers working with market risk and volatility prediction.

ADVANCED MODELS

Prophet (Developed by Facebook)

Facebook Prophet is an open-source forecasting tool designed for business time series with strong seasonality effects. It uses both curve fitting and custom seasonality modeling to create accurate forecasts with easy-to-understand parameters.

```
from prophet import Prophet
import pandas as pd

# Sample data: Date (ds) and Value (y)
df = pd.DataFrame({
    'ds': pd.date_range(start='2020-01-01', periods=365),
    'y': [100 + i + 10*(i%7) for i in range(365)] # Weekly seasonality
})

# Initialize and fit model
model = Prophet(
    yearly_seasonality=True,
    weekly_seasonality=True,
    daily_seasonality=False
)
model.fit(df)

# Create future dataframe (30-day forecast)
future = model.make_future_dataframe(periods=30)
```

```
forecast = model.predict(future)

# Plot results
fig = model.plot(forecast)
```

LSTM Networks

LSTM stands for Long Short-Term Memory. It is a special version of RNN designed to find long-term dependencies in information that is arranged in sequence. LSTMs address the issue of vanishing gradients by storing long-term information with the help of memory cells and gates.

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Generate synthetic univariate time series data
def generate_time_series(n=1000):
    time = np.arange(n)
    data = np.sin(0.1*time) + 0.1*np.random.randn(n)
    return data.reshape(-1, 1)

data = generate_time_series()

# Prepare sliding window samples
def create_dataset(data, window_size=20):
    X, y = [], []
    for i in range(len(data)-window_size):
        X.append(data[i:i+window_size])
        y.append(data[i+window_size])
    return np.array(X), np.array(y)

X, y = create_dataset(data)

# Build LSTM model
model = Sequential([
    LSTM(50, activation='tanh', input_shape=(20, 1)),
```

```
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=20, batch_size=32)

# Predict next value
last_window = data[-20:].reshape(1, 20, 1)
next_value = model.predict(last_window)
```

MODEL EVALUATION

Assessing the model you develop is necessary in machine learning. We need to test a trained model to ensure how well it handles new data. As a result, the model has a greater ability to notice important features in various types of inputs. The metrics used to measure a model's performance change depending on if the problem is regression, classification or clustering. We will now speak about metrics related to regression.

Performance Metrics

MAE or **Mean Absolute Error**, shows the average size of errors in predictions and ignores their direction. This means the mean of the absolute values of the differences between actual and predicted values is called MAE. It simplifies understanding, as it gives you an average of the errors in the original units. Better performance results from a low value of MAE in the model..

Formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where:

y_i is the real value.

$\hat{y}_i$ is the value that is predicted.

n is the total number of observations.

Example:

```
from sklearn.metrics import mean_absolute_error

# Sample data
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]

# MAE Calculation
mae = mean_absolute_error(y_true, y_pred)
print("Mean Absolute Error:", mae)
```

Mean Squared Error (MSE)

The Mean Squared Error (MSE) is another common performance metric. It calculates the average of the squared differences between the predicted and actual values. Unlike MAE, MSE gives more weight to larger errors because the errors are squared. This makes MSE sensitive to outliers. A lower MSE indicates a better fit of the model to the data.

Formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

y_i is the actual value.

$\hat{y}_i$ is the predicted value.

n is the total number of observations.

Example:

```
from sklearn.metrics import mean_squared_error

# MSE Calculation
mse = mean_squared_error(y_true, y_pred)
print("Mean Squared Error:", mse)
```

Root Mean Squared Error (RMSE)

The Root Mean Squared Error (RMSE) is the square root of the MSE. By taking the square root, RMSE brings the error metric back to the same unit as the original data, which makes it more interpretable than MSE. Like MSE, RMSE gives more importance to larger errors. It is often used when the goal is to penalize larger deviations more heavily.

Formula:

$$RMSE = \sqrt{MSE}$$

Example:

```
import numpy as np
# RMSE Calculation
rmse = np.sqrt(mse)
print("Root Mean Squared Error:", rmse)
```

Mean Absolute Percentage Error (MAPE)

The Mean Absolute Percentage Error (MAPE) is a relative error metric that expresses the accuracy of the model as a percentage. It calculates the average of the absolute percentage differences between the predicted and actual values. One advantage of MAPE is that it provides a clear idea of the model's performance in percentage terms, which can be more intuitive for comparison. However, MAPE can be problematic when actual values are close to zero, as it can lead to infinite or undefined values.

Formula:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

Example:

```
import numpy as np

# MAPE Calculation
mape = np.mean(np.abs((np.array(y_true) - np.array(y_pred)) / np.array(y_true))) * 100
print("Mean Absolute Percentage Error:", mape)
```

Cross-Validation for Time Series

Cross-validation is a powerful technique used to assess the performance of a machine learning model. However, when dealing with time series data, traditional cross-validation methods like K-fold cross-validation may not be appropriate because they do not account for the temporal order of the data. Time series data exhibits temporal dependencies, meaning the order of the data points matters,future data points cannot be used to predict past ones. For this reason, special cross-validation strategies have been developed for time series data to preserve the integrity of the temporal structure.

Here, we'll discuss a few methods used for cross-validation in time series forecasting:

1. Rolling/Expanding Window Cross-Validation

Rolling or expanding window cross-validation is one of the most common techniques used in time series data. Here, the model is trained on more information each time and tested on data from the following period.

Here, the training set always has the same size. Each step in the training process, one more point is left out and a new one comes in. This technique represents a situation where the amount of data you can use for predictions is known in advance.

Expanded Window: Here, new information is added to the training set over time. After folding the data, the training set gets

larger and the model is evaluated during the next period. The model assumes that the amount of available data increases with the passage of time.

An example of Rolling Window Cross-Validation is shown below:

```
import numpy as np
from sklearn.model_selection import TimeSeriesSplit
from sklearn.linear_model import LinearRegression

# Time Series Data
X = np.array([i for i in range(1, 21)]).reshape(-1, 1)  # Example feature
y = np.array([2*i + 1 for i in range(1, 21)])  # Example target

# Initialize the Rolling Window Cross-Validation with 5 splits
tscv = TimeSeriesSplit(n_splits=5)

# Create a model
model = LinearRegression()

# Perform the cross-validation
for train_index, test_index in tscv.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    print(f"Test indices: {test_index}, Predictions: {predictions}")
```

2. Walk-Forward Validation (Walk-Forward Testing)

Walk forward validation is similar to expanding window cross-validation, except you repeat the process by training with all the data you have and test it against the following observation. Every time a test is run, the model receives the new observations and the process continues.

You can use this method when you want to replicate a situation where predictions from the model rely on data from the past and get updated when more data is obtained.

Example of Walk-Forward Validation:

```
# Walk-forward validation (Expanding window)
for i in range(1, len(X)):
   X_train, X_test = X[:i], X[i:i+1]  # Train on all previous points, test on the next one
   y_train, y_test = y[:i], y[i:i+1]

   model.fit(X_train, y_train)
   prediction = model.predict(X_test)

   print(f"Test index: {i}, Prediction: {prediction}, True value: {y_test}")
```

3. Blocked Cross-Validation (TimeSeriesSplit that Set the Same Train-Test Partition)

The information is broken into separate time intervals and the model is taught with each interval before being tested on the next one. You may use this when you focus on model performance during periods that were not included in training. It assumes that the blocks are not closely related in terms of time.

Example of Blocked Cross-Validation:

```
from sklearn.model_selection import TimeSeriesSplit

# Use TimeSeriesSplit to simulate blocked cross-validation with 2 splits
tscv = TimeSeriesSplit(n_splits=2)

for train_index, test_index in tscv.split(X):
   X_train, X_test = X[train_index], X[test_index]
   y_train, y_test = y[train_index], y[test_index]

   model.fit(X_train, y_train)
   predictions = model.predict(X_test)
```

```
    print(f"Test indices: {test_index}, Predictions: {predictions}")
```

4. Leave-P-Out Cross-Validation (or TimeSeriesLeavePOut)

This way of performing cross-validation is a modification of the Leave-P-Out method, created for analyzing time series data. Part of the data (represented in this case as "P" time periods) is used for testing purposes, while the model is created based on the rest. Although calculating here is more time-consuming, this method allows you to check a model's accuracy in several time periods.

5. TimeSeriesSplit in Scikit-Learn

The TimeSeriesSplit function in Scikit-learn carries out time series cross-validation. Data are divided into k groups so that the original order of the observations is maintained. Each time period in the test set is separated from data included in the training set.. It ensures that data leakage does not occur by using future observations to predict past ones.

Example:

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.linear_model import LinearRegression

# TimeSeriesSplit with 3 splits
tscv = TimeSeriesSplit(n_splits=3)

# Create the model
model = LinearRegression()

# Perform cross-validation
for train_index, test_index in tscv.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    model.fit(X_train, y_train)
```

```
predictions = model.predict(X_test)

print(f"Predictions: {predictions}, Actual values: {y_test}")
```

REAL-WORLD APPLICATION

we will explore three real-world applications of time series forecasting, detailing the models used for each and providing code implementations for stock price prediction, sales forecasting, and weather forecasting.

Stock Price Prediction

Problem Overview: Stock price prediction is a classic time series forecasting problem where the goal is to predict future stock prices based on historical data. Stock prices are influenced by multiple factors, and various models like ARIMA (AutoRegressive Integrated Moving Average) and LSTM (Long Short-Term Memory) can be used to capture the underlying patterns in the data.

Models Used:

ARIMA: A statistical model used to predict future values based on past data by capturing the temporal dependencies.

LSTM: LSTM belongs to the type of RNN called Recurrent Neural Networks, adept at learning from trends across several time points which is why it's commonly used for predicting stock prices.

Code Implementation:

ARIMA Model:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
```

```
# Load historical stock price data (e.g., from Yahoo Finance)
data = pd.read_csv('stock_prices.csv', index_col='Date', parse_dates=True)

# Use the 'Close' price for prediction
stock_data = data['Close']

# Split data into train and test sets
train_size = int(len(stock_data) * 0.8)
train, test = stock_data[:train_size], stock_data[train_size:]

# Fit the ARIMA model (order=(5,1,0) is just an example, should be tuned)
model = ARIMA(train, order=(5, 1, 0))
model_fit = model.fit()

# Make predictions
predictions = model_fit.forecast(steps=len(test))

# Plot the results
plt.plot(test.index, test, label='Actual')
plt.plot(test.index, predictions, label='Predicted')
plt.legend()
plt.show()

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(test, predictions))
print(f"RMSE: {rmse}")
```

LSTM Model:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Load historical stock price data
data = pd.read_csv('stock_prices.csv', index_col='Date', parse_dates=True)
stock_data = data['Close']
```

```
# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
stock_data_scaled = scaler.fit_transform(stock_data.values.reshape(-1, 1))

# Prepare the data for LSTM (creating a sliding window of data)
def create_dataset(data, time_step=60):
    X, y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

time_step = 60
X, y = create_dataset(stock_data_scaled, time_step)

# Split the data into train and test sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Reshape the data for LSTM [samples, time_steps, features]
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

# Build the LSTM model
model = Sequential()
model.add(LSTM(units=50,                                    return_sequences=True,
input_shape=(X_train.shape[1], 1)))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dense(units=1))

# Compile and fit the model
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=10, batch_size=32)

# Make predictions
predictions = model.predict(X_test)
```

```
# Inverse transform the predictions
predictions = scaler.inverse_transform(predictions)

# Plot the results
plt.plot(scaler.inverse_transform(y_test.reshape(-1, 1)), label='Actual')
plt.plot(predictions, label='Predicted')
plt.legend()
plt.show()
```

Sales Forecasting

Problem Overview: Time series forecasting plays an important role in sales forecasting. To do all these things well, businesses have to forecast sales accurately. Seasonal patterns and trends can be modelled well with Exponential Smoothing and Prophet, so these are common choices for forecasting sales.

Models Used:

Exponential Smoothing: This approach provides the highest significance to the latest data and often helps in predicting sales.

Prophet: a Facebook tool used for forecasting results that can include daily, weekly and yearly changes, holidays and different events.

Code Implementation:

Exponential Smoothing Model:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Load sales data (e.g., monthly sales)
data = pd.read_csv('sales_data.csv', index_col='Date', parse_dates=True)

# Train-Test Split
train_size = int(len(data) * 0.8)
train, test = data[:train_size], data[train_size:]
```

```
# Fit the Exponential Smoothing model
model = ExponentialSmoothing(train['Sales'], trend='add', seasonal='add', seasonal_periods=12)
model_fit = model.fit()

# Make predictions
predictions = model_fit.forecast(steps=len(test))

# Plot the results
plt.plot(test.index, test['Sales'], label='Actual')
plt.plot(test.index, predictions, label='Predicted')
plt.legend()
plt.show()
```

Prophet Model:

```
import pandas as pd
from fbprophet import Prophet
import matplotlib.pyplot as plt

# Load sales data
data = pd.read_csv('sales_data.csv')
data = data.rename(columns={'Date': 'ds', 'Sales': 'y'})  # Prophet requires 'ds' and 'y' columns

# Create and fit the Prophet model
model = Prophet()
model.fit(data)

# Make future predictions (e.g., for 12 months ahead)
future = model.make_future_dataframe(data, periods=12, freq='M')
forecast = model.predict(future)

# Plot the forecast
model.plot(forecast)
plt.show()
```

Weather Forecasting

Problem Overview: Weather Forecasting involves predicting what the weather will do by looking back at history. SARIMA and LSTM models are suitable for finding seasonal and long-term trends in the data about weather..

Models Used:

SARIMA: Seasonal ARIMA, a modification of the ARIMA model, is designed to handle seasonal data, which is crucial in weather forecasting.

LSTM: A deep learning model capable of capturing complex non-linear relationships in time series data, making it suitable for weather prediction.

Code Implementation:

SARIMA Model:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Load weather data (e.g., monthly temperature)
data = pd.read_csv('weather_data.csv', index_col='Date', parse_dates=True)

# Train-Test Split
train_size = int(len(data) * 0.8)
train, test = data[:train_size], data[train_size:]

# Fit the SARIMA model
model = SARIMAX(train['Temperature'], order=(1, 1, 1), seasonal_order=(1,
1, 1, 12))
model_fit = model.fit()

# Make predictions
predictions = model_fit.forecast(steps=len(test))

# Plot the results
```

```
plt.plot(test.index, test['Temperature'], label='Actual')
plt.plot(test.index, predictions, label='Predicted')
plt.legend()
plt.show()
```

LSTM Model:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Load weather data
data = pd.read_csv('weather_data.csv', index_col='Date', parse_dates=True)
weather_data = data['Temperature']

# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
weather_data_scaled = scaler.fit_transform(weather_data.values.reshape(-1, 1))

# Prepare the data for LSTM
def create_dataset(data, time_step=60):
    X, y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

time_step = 60
X, y = create_dataset(weather_data_scaled, time_step)

# Split the data into train and test sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Reshape the data for LSTM
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
```

```
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

# Build the LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True,
input_shape=(X_train.shape[1], 1)))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dense(units=1))

# Compile and fit the model
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=10, batch_size=32)

# Make predictions
predictions = model.predict(X_test)

# Inverse transform the predictions
predictions = scaler.inverse_transform(predictions)

# Plot the results
plt.plot(scaler.inverse_transform(y_test.reshape(-1, 1)), label='Actual')
plt.plot(predictions, label='Predicted')
plt.legend()
plt.show()
```

QUESTIONS

1. You are given a time series dataset with daily temperature data. What steps would you follow to forecast the temperature for the next week using LSTM?
2. What is an ARIMA model, and what does each of its components (p, d, q) represent?
3. What is the purpose of differencing in time series modeling?
4. How does the SARIMA model extend the ARIMA model?
5. Explain the difference between an AR (AutoRegressive) model and an MA (Moving Average) model in time series analysis.
6. You have a dataset with monthly sales data, and you are asked to fit an ARIMA model. Describe the steps you would follow to build the model.
7. How would you choose the optimal seasonal period (s) for a SARIMA model?
8. Given a dataset with daily temperature data, how would you use an ARIMA model to forecast the temperature for the next 30 days?
9. What is the purpose of using a "grid search" in tuning the parameters of a time series model, such as ARIMA or SARIMA?
10. You are given a time series with significant seasonality. How would you incorporate seasonality into your model, and which model would you use?

MODULE 17

MACHINE LEARNING WITH PYTHON

This module outlines the main features of machine learning which is a fundamental area of AI that allows systems to learn and enhance their functions over time, mostly by themselves. You will learn important ideas and methods for resolving regression, classification and clustering issues in Python.

After completing this module, you will learn to:

- Understand the basic principles of machine learning.
- Implement regression, classification, and clustering algorithms in Python.
- Apply machine learning models to real-world datasets.
- Evaluate and interpret model performance.

INTRODUCTION

Machine learning is a branch of artificial intelligence concerned with making systems capable of understanding data and making choices or predictions. You have to guide your computer to stop being lazy and instead start using its own brain. Rather than explaining everything, you provide it with lots of data and hope it discovers how to do what you want it to do. It is a creative AI field in which computers study patterns, guess what might happen or occasionally choose something surprising.

Machine learning doesn't rely on being programmed to draw conclusions from sets of data. In general, it falls into two main areas: **supervised learning and unsupervised learning.**

SUPERVISED VERSUS UNSUPERVISED LEARNING

Supervised Learning

In supervised learning, a model is taught by giving it certain data sets along with the expected outputs.

In supervised learning, a student is taught by a trustworthy mentor who already knows the answers. Here, the model is built by training it on data sets where everyone knows what the correct outcome should be. Training the model is aimed at showing it how the inputs influence the outputs, so it can accurately predict when presented with unknown data.

So, let's say you are training a model to identify cats in images. Giving the computer thousands of labeled photos of cats allows it to discover the characteristics that differentiate cats from other things. We often use supervised learning for tasks including regression, where prices are estimated and classification, where the label will be spam or not spam. The idea is to fit a boundary through the data, one that is as smart as possible thanks to how the model was trained.

Many supervised learning models are available, designed to solve distinct kinds of problems using different types of data. A few of the top supervised learning models include:

1. **Linear Regression** – Used to find a straight line that best fits the given data to help predict certain values.. It assumes a linear relationship between input features and the target.
   Example Use Case: Predicting house prices based on square footage.

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Sample data
X = np.array([[1000], [1500], [2000], [2500]])  # Square footage
y = np.array([200000, 250000, 300000, 350000])  # Prices
```

```
model = LinearRegression()
model.fit(X, y)

# Predict price for 1800 sq ft
predicted_price = model.predict([[1800]])
print(f"Predicted Price: ${predicted_price[0]:,.2f}")
```

2. **Logistic Regression** – Despite its name, it's used for classification tasks. It estimates the probability that an instance belongs to a particular class, making it great for binary classification like spam detection.
   Example Use Case: Predicting whether an email is spam or not.

```
from sklearn.linear_model import LogisticRegression

# Features: [has_link, num_exclamations]
X = [[1, 5], [0, 0], [1, 10], [0, 1]]
y = [1, 0, 1, 0]  # 1: spam, 0: not spam

model = LogisticRegression()
model.fit(X, y)

# Predict for a new email
prediction = model.predict([[1, 3]])
print("Spam" if prediction[0] == 1 else "Not Spam")
```

3. **Decision Trees** – These models split the data into branches based on feature values, like a flowchart. They're easy to understand and interpret.
   Example Use Case: Deciding loan approval based on income and credit score.

```
from sklearn.tree import DecisionTreeClassifier

# Features: [income, credit_score]
X = [[50000, 600], [80000, 700], [30000, 500], [90000, 800]]
y = [0, 1, 0, 1]  # 1: Approved, 0: Not Approved
```

```
model = DecisionTreeClassifier()
model.fit(X, y)

# Predict for a new applicant
prediction = model.predict([[75000, 650]])
print("Loan Approved" if prediction[0] == 1 else "Loan Denied")
```

4. **Random Forest** – An ensemble of decision trees that improves accuracy by averaging multiple trees to reduce overfitting.
   Example Use Case: Predicting whether a customer will churn.

```
from sklearn.ensemble import RandomForestClassifier
# Features: [tenure_months, monthly_charges]
X = [[2, 80], [12, 45], [24, 60], [1, 95]]
y = [1, 0, 0, 1]  # 1: Churned, 0: Stayed
model = RandomForestClassifier(n_estimators=100)
model.fit(X, y)
prediction = model.predict([[6, 70]])
print("Churn" if prediction[0] == 1 else "Stay")
```

5. **Support Vector Machines (SVM)** – These models find the best boundary (or hyperplane) that separates different classes. They're effective in high-dimensional spaces.
   Example Use Case: Classifying if a tumor is benign or malignant.

```
from sklearn.svm import SVC

# Features: [size, smoothness]
X = [[1.0, 0.3], [1.2, 0.2], [2.5, 0.8], [3.0, 1.0]]
y = [0, 0, 1, 1]  # 0: Benign, 1: Malignant

model = SVC()
model.fit(X, y)

prediction = model.predict([[2.0, 0.6]])
print("Malignant" if prediction[0] == 1 else "Benign")
```

6. **K-Nearest Neighbors (KNN)** – KNN classifies new data points based on the majority class among its K closest neighbors in the feature space that is, this lazy learner makes predictions based on the majority vote of its nearest neighbors. It's simple and works well with small datasets.
   Example Use Case: Classifying a new flower species based on petal length and width.

```
from sklearn.neighbors import KNeighborsClassifier

# Features: [petal_length, petal_width]
X = [[1.4, 0.2], [4.7, 1.4], [5.0, 1.5], [1.3, 0.2]]
y = [0, 1, 1, 0]  # 0: Setosa, 1: Versicolor

model = KNeighborsClassifier(n_neighbors=3)
model.fit(X, y)

prediction = model.predict([[1.5, 0.3]])
print("Setosa" if prediction[0] == 0 else "Versicolor")
```

7. **Naive Bayes** – Based on Bayes' theorem, this model assumes independence among features and is particularly good for text classification tasks.
   Example Use Case: Classifying movie reviews as positive or negative.

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

corpus = ["great movie", "bad movie", "amazing plot", "terrible acting"]
y = [1, 0, 1, 0]  # 1: Positive, 0: Negative

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

model = MultinomialNB()
model.fit(X, y)
```

```
new_review = vectorizer.transform(["amazing acting"])
prediction = model.predict(new_review)
print("Positive" if prediction[0] == 1 else "Negative")
```

8. **Neural Networks** – Inspired by the human brain, these models consist of layers of interconnected nodes and are powerful for handling complex patterns and large datasets.
   Example of a typical Use Case: Interpreting written digits (from the MNIST data).

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target)

model = MLPClassifier(hidden_layer_sizes=(64,), max_iter=1000)
model.fit(X_train, y_train)

score = model.score(X_test, y_test)
print(f"Accuracy: {score:.2f}")
```

Unsupervised Learning

During unsupervised learning, the model is trained using input data that does not have any labels. When you apply unsupervised learning, your computer is left to explore a new place all by itself. This learning method receives unlabeled data, so the teacher doesn't define what the outputs are. The objective is to uncover patterns, groupings or structures that are present in the data.
Common tasks in unsupervised learning include:

- **Clustering**: Grouping similar data points together.
- **Dimensionality Reduction**: Reducing the number of features while preserving important information.

1. K-Means Clustering

K-Means is a popular clustering algorithm that groups data into **K clusters** based on feature similarity. It works by initializing K centroids, assigning data points to the nearest centroid, and updating the centroids iteratively.

Example Use Case: Customer segmentation for targeted marketing.

```
from sklearn.cluster import KMeans
import numpy as np

# Sample data: [annual_income, spending_score]
X = np.array([[15, 39], [16, 81], [17, 6], [18, 77], [19, 40], [20, 76]])

model = KMeans(n_clusters=2, random_state=0)
model.fit(X)

print("Cluster labels:", model.labels_)
print("Centroids:", model.cluster_centers_)
```

2. Hierarchical Clustering (Agglomerative Clustering)

Hierarchical clustering builds a hierarchy of clusters either from the bottom up (agglomerative) or top down (divisive). It does not require you to pre-specify the number of clusters.

Example Use Case: Creating a dendrogram of gene similarities in bioinformatics.

```
from sklearn.cluster import AgglomerativeClustering

X = [[1, 2], [2, 3], [5, 6], [6, 7]]

model = AgglomerativeClustering(n_clusters=2)
labels = model.fit_predict(X)

print("Cluster labels:", labels)
```

3. DBSCAN (Density-Based Spatial Clustering)

DBSCAN groups points that are closely packed together and marks points in low-density regions as outliers. It is useful for discovering clusters of arbitrary shape and handling noise.

Example Use Case: Identifying unusual patterns in network traffic (intrusion detection).

```
from sklearn.cluster import DBSCAN

X = [[1, 2], [2, 2], [2, 3], [8, 7], [8, 8], [25, 80]]

model = DBSCAN(eps=3, min_samples=2)
labels = model.fit_predict(X)

print("Cluster labels:", labels)  # -1 indicates noise/outliers
```

4. Principal Component Analysis (PCA)

PCA is a **dimensionality reduction** technique. It transforms data into a new coordinate system to highlight the most important information using fewer features. PCA is often used before clustering or visualization.

Example Use Case: Visualizing high-dimensional data like face images or gene expressions.

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

iris = load_iris()
X = iris.data

pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)

plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=iris.target)
plt.title("PCA of Iris Dataset")
```

```
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.show()
```

5. t-SNE (t-distributed Stochastic Neighbor Embedding)

t-SNE is another dimensionality reduction technique, often used for visualizing high-dimensional data in 2D or 3D. Unlike PCA, it focuses on preserving local structure.

Example Use Case: Visualizing handwritten digits or document clusters.

```
from sklearn.manifold import TSNE
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt

digits = load_digits()
X = digits.data
y = digits.target

tsne = TSNE(n_components=2, random_state=42)
X_embedded = tsne.fit_transform(X)

plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=y, cmap='tab10')
plt.colorbar()
plt.title("t-SNE Visualization of Digits Dataset")
plt.show()
```

Performance Metrics in Machine Learning

In machine learning, evaluating the performance of a model is crucial for determining how well it makes predictions. **Performance metrics** provide quantitative measures of how well a model is performing, enabling us to compare different models or fine-tune a model to improve its accuracy.

Performance Metrics for Classification Models

1. Accuracy

Accuracy is the most straightforward performance metric. It is the ratio of the number of correct predictions to the total number of predictions made.

Formula:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} * 100$$

where:

- TP = True Positives (correctly predicted positives)
- TN = True Negatives (correctly predicted negatives)
- FP = False Positives (incorrectly predicted positives)
- FN = False Negatives (incorrectly predicted negatives)

We will talk about TP, TN, FP and FN shortly in tis module.

Use Case: It's useful when the class distribution is relatively balanced.

Example:

```
from sklearn.metrics import accuracy_score

# Example data (predictions and true labels)
y_true = [1, 0, 1, 1, 0]
y_pred = [1, 0, 1, 0, 0]

accuracy = accuracy_score(y_true, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

2. Precision

Precision tells you how many of the items your model identified as positive are actually positive. It's particularly important in imbalanced datasets, where false positives are costly.

Formula:

$$Precision = \frac{TP}{TP + FP} * 100$$

Use Case: Useful when the cost of false positives is high (e.g., diagnosing a disease).

Example:

```
from sklearn.metrics import precision_score

precision = precision_score(y_true, y_pred)
print(f"Precision: {precision:.2f}")
```

3. Recall (Sensitivity or True Positive Rate)

Recall measures how many actual positive instances were identified correctly by the model. It tells you how good your model is at catching the positive cases.

Formula:

$$Recall = \frac{TP}{TP + FN} * 100$$

Use Case: Important when false negatives are costly (e.g., failing to detect fraud).

Example:

```
from sklearn.metrics import recall_score

recall = recall_score(y_true, y_pred)
print(f"Recall: {recall:.2f}")
```

4. F1-Score

F1-score is the harmonic mean of Precision and Recall. It balances both the precision and recall, and is particularly useful when you have an uneven class distribution.

Formula:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Example:

```
from sklearn.metrics import f1_score
f1 = f1_score(y_true, y_pred)
print(f"F1-Score: {f1:.2f}")
```

5. ROC Curve and AUC (Area Under the Curve)

The **ROC Curve** plots the true positive rate (recall) against the false positive rate while **AUC** is the area under this curve, indicating how well the model distinguishes between classes.

Example:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Sample data
y_prob = [0.9, 0.1, 0.8, 0.4, 0.5] # predicted probabilities for class 1

fpr, tpr, _ = roc_curve(y_true, y_prob)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)'
% roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
```

```
plt.legend(loc='lower right')
plt.show()
print(f"AUC: {roc_auc:.2f}")
```

Performance Metrics for Regression Models

1. Mean Absolute Error (MAE)

MAE calculates the average of the absolute errors between predicted and actual values. It's easy to interpret because it represents the average difference in the units of the target.

Formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where:

y_i is the actual value.

$\hat{y}_i$ is the predicted value.

n is the total number of observations.

Example:

```
from sklearn.metrics import mean_absolute_error

# Example data (actual vs predicted)
y_true = [100, 200, 300, 400, 500]
y_pred = [110, 210, 290, 395, 505]

mae = mean_absolute_error(y_true, y_pred)
print(f"Mean Absolute Error: {mae}")
```

2. Mean Squared Error (MSE)

MSE computes the average of the squared differences between the predicted and actual values. It penalizes larger errors more heavily than MAE.

Formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Example:

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_true, y_pred)
print(f"Mean Squared Error: {mse}")
```

3. R-squared (R^2)

R^2 measures the proportion of the variance in the target variable that is explained by the model. If the R^2 is closer to 1, the model fits the data better.

Formula:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Where

SS_{res} = **Sum of squared residual (errors)** $\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

SS_{tot} = **Total sum of squares** $\sum_{i=1}^{n}(y_i - \bar{y})^2$

$\bar{y}$ = **mean of actual y values.**

Example:

```
from sklearn.metrics import r2_score

r2 = r2_score(y_true, y_pred)
print(f"R-squared: {r2}")
```

Confusion Matrix:

A **Confusion Matrix** allows you to measure the accuracy of a machine learning classifier. It helps a lot when analyzing the outcomes of a classification algorithm. The matrix lets you know

the percentage of correct predictions and the percentage of incorrect ones made by your model.
Usually, a confusion matrix is set up in this order:

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Terms in the Confusion Matrix:

1. **True Positive (TP)**: The instances that were correctly identified as positive.
2. **False Positive (FP)**: The number of cases that are truly negative but were predicted to be positive.
3. **True Negative (TN)**: The amount of negative instances successfully identified as negative by the model.
4. **False Negative (FN)**: The amount of positive outcomes that were wrongly foreseen as negative..

Example:
Let's imagine that your confusion matrix is as follows:

	Predicted Positive	**Predicted Negative**
Actual Positive	50	10
Actual Negative	5	100

Let's compute some of the metrics:

1. **Accuracy**:

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score, accuracy_score

# True values (Actuals) and Predicted values
y_true = [1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1]
y_pred = [1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0]

# Confusion Matrix
```

```
cm = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:")
print(cm)

# Extracting the values from the confusion matrix
TP = cm[1, 1] # True Positive
TN = cm[0, 0] # True Negative
FP = cm[0, 1] # False Positive
FN = cm[1, 0] # False Negative

# Calculating performance metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
```

QUESTIONS

1. What is machine learning and how is it different from traditional programming?
2. What is the difference between supervised and unsupervised learning? Provide examples.
3. Given this code, identify if it's supervised or unsupervised and justify your answer:

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3)
model.fit(X)
```

4. In supervised learning, what is the purpose of the label in training data?
5. What does the label y represent in this code?

```
X = [[50000, 600], [80000, 700]]
y = [0, 1]
```

6. Which supervised learning algorithm would you use to predict continuous values?
7. What is the expected output type of this model?

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

8. Write a Python script to train a linear regression model on a dataset.
9. Use K-Means clustering to group the Iris dataset into 3 clusters.
10. What is the difference between linear regression and logistic regression?
11. Train a linear regression model on the Boston Housing dataset and evaluate its performance.
12. Use logistic regression to classify the Iris dataset into two classes (setosa vs non-setosa).

13. What is the difference between decision trees and random forests?
14. Train a decision tree classifier on the Titanic dataset and evaluate its performance.
15. Use SVM to classify the Iris dataset into three classes.
16. What is the difference between K-Means and hierarchical clustering?
17. Use K-Means clustering to group the Iris dataset into 3 clusters.
18. Perform hierarchical clustering on the Wine dataset and visualize the dendrogram.
19. Explain the difference between Precision and Recall. In which scenarios would you prioritize one over the other?
20. What is the F1-score, and why is it considered a more balanced metric compared to Accuracy? Provide an example where using F1-score would be more beneficial than using accuracy.
21. Given the following confusion matrix, calculate the Precision, Recall, and F1-Score:

	Predicted Positive	Predicted Negative
Actual Positive	50	10
Actual Negative	5	100

22. You have a regression model and your R-squared value is 0.85. What does this imply about the model's performance, and what would be the impact of a low R-squared value?
23. Write a Python function to compute the Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared for a given set of true and predicted values. Provide an example with a dataset.
24. Given the following confusion matrix for a binary classification problem:

	Predicted Positive	Predicted Negative
Actual Positive	120	30
Actual Negative	25	150

25. Calculate the following performance metrics:
    a) Accuracy
    b) Precision
    c) Recall
    d) F1-Score
26. You have a confusion matrix for a spam detection model as follows:

	Predicted Spam	Predicted Not Spam
Actual Spam	90	10
Actual Not Spam	5	95

    a) What is the precision and recall of the model?
    b) What is the F1-Score of the model?
    c) What does a high precision and recall indicate about the model's performance?

MODULE 18

REAL-WORLD DATA SCIENCE PROJECTS

This module focuses on applying the concepts and techniques learned in previous modules to real-world data science projects. It covers the end-to-end process of a data science project and provides case studies to demonstrate practical applications.

END-TO-END DATA SCIENCE PROJECT

A data science project typically follows a structured workflow, from problem definition to deployment. This module walks you through each step of the process.

1. Problem Definition

The first step in any data science project is to clearly define the problem you are trying to solve. This involves understanding the business context, identifying the goal, and defining success metrics. Think of it like preparing for a Nigerian wedding. Before you start shopping for aso ebi or hiring the DJ, you need to know *who* is getting married, *when*, and *what* kind of party they want. In data science, rushing in without a well-defined problem is like trying to cook jollof rice without knowing whether your guests prefer it spicy or mild. You might end up with a dish nobody wants. So always begin with a clear understanding of the problem. It saves time, energy, and a lot of debugging headaches later on.

Problem: Predicting Student Dropout in a Polytechnic

In many Nigerian polytechnics, student dropout rates pose a serious challenge to educational progress and institutional reputation. Understanding the underlying reasons for these

dropouts is essential to formulating effective solutions. In this context, the management of a Nigerian polytechnic has expressed concern over the increasing number of students who abandon their studies before completing their programs. The institution believes that leveraging data science can provide a strategic solution by uncovering patterns and risk factors that may contribute to student attrition.

The core problem is whether it's possible to develop a machine learning model that can predict which students are likely to drop out. You should analyze aspects such as a person's grades, their attendance, payment of fees and their involvement in activities on campus. An early-identification method for at-risk students is being created by studying past student records.

The project intends to create a model that helps flag students who are most likely to drop out during their first academic year. If the model is effective, it might be utilized as a support tool in both academic and counseling departments. Early mentoring, financial support or academic help ensure students retain their interest in education, so their academic achievements also improve.

2. Data Collection

Gathering data is important in any data science task, as it supports the accuracy and relevance of your model. This means you must collect and record data that is linked to your problem. This means extracting information from various databases inside the school, including student information and management information systems at Nigerian polytechnics.

Other useful data is obtained from education platforms, forums for scholars and students or from the Nigerian Bureau of Statistics. Referring to student dropout, data used would be their grades, their class attendance, how fees are paid and involvement in social activities. If data is collected correctly, the model will have all the

information it needs to identify significant trends and predict accurately.

```
import pandas as pd
# Load data from a CSV file
data = pd.read_csv("student_performance.csv")
print(data.head())
```

3. Data Cleaning

Preparing data for analysis and modeling begins with data cleaning. In many practical situations, mainly in Nigerian polytechnics, some student information may be missing, duplicated or have incorrect forms. When not addressed such issues can result in faulty training and predictions by the model.

After gathering the data for dropout prediction, you should inspect for and treat missing academic scores, adjust inconsistent forms of categorical variables and eliminate records when key features are completely blank. Cleaning the data correctly is important for making sure it can be used effectively for machine learning.

Sample Code for Data Cleaning (Dropout Prediction Scenario)

```
import pandas as pd
import numpy as np

# Sample dataset
data = {
   'student_id': [101, 102, 103, 104, 104],
   'attendance_rate': [0.95, 0.60, np.nan, 0.80, 0.80],
   'gpa': [3.5, 2.1, 2.8, 3.2, 3.2],
   'financial_aid': ['Yes', 'No', 'YES', np.nan, 'Yes'],
   'dropout': [0, 1, 0, 1, 1]
}
```

```
df = pd.DataFrame(data)

print("Original Data:")
print(df)

# Remove duplicate student entries
df = df.drop_duplicates(subset='student_id')

# Handle missing values
df['attendance_rate'].fillna(df['attendance_rate'].mean(), inplace=True)
df['financial_aid'].fillna('Unknown', inplace=True)

# Standardize categorical values
df['financial_aid'] = df['financial_aid'].str.lower()

print("\nCleaned Data:")
print(df)
```

4. Model Building

After preparing the data, the next phase is to develop a model. In this phase, a machine learning system studies the past data so that it can make predictions about things it has not encountered before. When building a model, one should pick the features, set the target variable, divide the data into training and test parts and decide on the best algorithm considering the problem.

In trying to predict dropouts, our features are attendance rate, GPA and financial aid status and the target variable is whether a student left school or stayed (binary classification: 1 = left school, 0 = stayed).

Sample Code: Model Building for Dropout Prediction

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.metrics import accuracy_score, classification_report

# Assume the cleaned dataset
data = {
    'attendance_rate': [0.95, 0.60, 0.78, 0.80],
    'gpa': [3.5, 2.1, 2.8, 3.2],
    'financial_aid': ['yes', 'no', 'yes', 'unknown'],
    'dropout': [0, 1, 0, 1]
}
df = pd.DataFrame(data)

# Encode categorical variable
le = LabelEncoder()
df['financial_aid'] = le.fit_transform(df['financial_aid'])

# Features and target
X = df[['attendance_rate', 'gpa', 'financial_aid']]
y = df['dropout']

# Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Build and train the model
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

5. Evaluation

Once the model is ready, it should be evaluated to confirm that it can perform correctly on new data. At this point, statistics are used to see how much the model is able to predict correctly on the

test set. In most cases, people use metrics such as accuracy, precision, recall and F1-score to evaluate classification problems like student dropout prediction and they are as follow:

- **Accuracy**: The proportion of correctly predicted instances out of all predictions.
- **Precision**: The proportion of true positive predictions out of all predicted positives.
- **Recall (Sensitivity)**: The proportion of true positive predictions out of all actual positives.
- **F1-score**: The harmonic mean of precision and recall, offering a balance between the two.

These metrics help us understand not just how many predictions the model got right, but also how useful and reliable those predictions are, especially important when dealing with imbalanced classes, such as when far fewer students drop out than remain.

Evaluation in Practice (Using Our Dropout Prediction Example)

We can now check the model's performance by applying sklearn's methods for model evaluation.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report

# Predictions from our trained model (assumed done already)
y_pred = model.predict(X_test)

# Evaluate using standard metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

```
# Print evaluation results
print("Evaluation Metrics:")
print(f"Accuracy:  {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall:    {recall:.2f}")
print(f"F1 Score:  {f1:.2f}")

# Display classification report
print("\nDetailed Classification Report:")
print(classification_report(y_test, y_pred))

# Confusion Matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Output

```
Evaluation Metrics:
Accuracy:  0.75
Precision: 0.67
Recall:    1.00
F1 Score:  0.80

Detailed Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.50      0.67         2
           1       0.67      1.00      0.80         1

Confusion Matrix:
[[1 1]
 [0 1]]
```

6. Deployment

After finishing training and testing the machine learning model, the following stage is to make it useful by deploying it. This means school principals or student support staff can rely on the

model to provide extra support to students at risk of dropping out early in their studies. It usually requires changing the model to provide a service such as through a web application or API.

Key Steps in Model Deployment

1. **Save the Trained Model**
   Use a serialization library like joblib or pickle to store the model.

```
import joblib

# Save the trained model to a file
joblib.dump(model, 'dropout_predictor_model.pkl')
```

2. **Create a RESTful API**
   You can use **Flask** or **FastAPI** to build an endpoint where new student data can be sent for prediction.

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

# Load the model
model = joblib.load('dropout_predictor_model.pkl')

# Initialize Flask app
app = Flask(__name__)

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    # Example input: {"features": [0.85, 1, 2, 0]}
    features = np.array(data['features']).reshape(1, -1)
    prediction = model.predict(features)
    return jsonify({'dropout_risk': int(prediction[0])})

if __name__ == '__main__':
    app.run(debug=True)
```

3. Using Docker

To containerize your app with the provided Dockerfile, follow these steps:

Step 1: Create the Dockerfile:

Save the following content into a file named Dockerfile (with no file extension):

```
# Use an official Python runtime as a parent image
FROM python:3.10

# Set the working directory in the container to /app
WORKDIR /app

# Copy the current directory contents (the app) into the container at /app
COPY . /app

# Install the required dependencies
RUN pip install flask joblib numpy

# Specify the command to run your app when the container starts
CMD ["python", "app.py"]
```

Step 2. Create Your Application (app.py):

Ensure that you have an app.py file in the same directory as the Dockerfile. The app.py file should contain your Flask application, for example:

```
from flask import Flask
import joblib
import numpy as np

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Dockerized Flask App!"

if __name__ == '__main__':
```

```
app.run(debug=True, host='0.0.0.0')
```

This is a basic Flask app for demonstration purposes.

Step 3. Build the Docker Image:

In the same directory as the Dockerfile and app.py, open your terminal and run the following command to build the Docker image:

```
docker build -t flask-app .
```

This command creates an image that can be used as a flask-app.

Step 4. Run the Docker Container:

Once the image is built, run the container with:

```
docker run -p 5000:5000 flask-app
```

This will run the Flask app inside the container and map port 5000 on your local machine to port 5000 in the container.

Step 5. Access the Application:

Now You can use your web browser to connect to http://localhost:5000.

4. **Deploy to the Cloud or Local Server**
   You can host the application on platforms like:
   - **Heroku**
   - **Render**
   - **AWS EC2**
   - **Azure Web Apps**
   - **Local School Server**
5. **Integrate with Existing Systems**
   - Connect the API to a school's **Student Information System (SIS)**.
   - Build a dashboard with **Streamlit** or **Dash** to visualize results and alerts.

- Send predictions to counselors for timely interventions.

Outcome after deployment

Imagine the registrar of a Nigerian polytechnic logs into a portal and uploads current student data. Behind the scenes, the deployed model evaluates each student's risk of dropping out. A report is generated with a list of at-risk students, complete with risk scores and suggestions for counseling or financial aid. This automated system helps improve student retention in a proactive, data-driven way.

FEATURE ENGINEERING IN DATA SCIENCE

Feature engineering is the process of transforming raw data into meaningful features that improve the performance of machine learning models. It includes techniques like feature selection, extraction, and transformation to enhance model accuracy and efficiency.

Steps in Feature Engineering

1. **Feature Creation**: Generating new features from existing ones.
2. **Feature Transformation**: Applying mathematical or statistical transformations (e.g., normalization, log scaling).
3. **Feature Selection**: Choosing the most relevant features to improve model efficiency.
4. **Feature Extraction**: Deriving new features from raw data (e.g., PCA, word embeddings).

Lets focus on Feature Selection

Feature Selection

Feature Selection in Machine Learning Feature selection is a crucial step in machine learning that removes redundant or irrelevant features, leading to:

- Redice overfitting
- Improved accuracy
- Faster training

  Feature selection is divided into two main types:
- **Single Feature Selection** (Individual selection methods)
- **Ensemble Feature Selection** (Combining multiple selection methods)

Single Feature Selection Methods

(a) Filter Methods These methods score features independently of the model.

Examples:

- **Chi-Square Test** (For categorical data)
- **Information Gain** (Mutual information)
- **Correlation Coefficient** (Pearson's, Spearman's)

Example Code for Chi-Square Test

```
from sklearn.feature_selection import SelectKBest, chi2
import pandas as pd
import numpy as np
# Sample dataset
X = np.random.randint(0, 50, (10, 5))  # 10 samples, 5 features
y = np.random.randint(0, 2, 10)  # Binary target

# Apply Chi-Square test
chi2_selector = SelectKBest(score_func=chi2, k=3)
X_selected = chi2_selector.fit_transform(X, y)

print("Selected Features (Chi-Square):", X_selected.shape[1])
```

(b) Wrapper Methods

Wrapper methods select features based on model performance.

Examples:

- **Recursive Feature Elimination (RFE)**
- **Forward Feature Selection**
- **Backward Elimination**

Example Code for RFE with Decision Tree

```
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier

# Define model
model = DecisionTreeClassifier()
rfe = RFE(model, n_features_to_select=3)

# Fit and transform data
X_selected = rfe.fit_transform(X, y)
print("Selected Features (RFE):", X_selected.shape[1])
```

(c) Embedded Methods These methods use built-in feature selection during model training. Examples:

- **LASSO Regression** (L1 regularization)
- **Decision Tree Feature Importance**

Example Code for Feature Importance in Random Forest

```
from sklearn.ensemble import RandomForestClassifier
# Train a RandomForest model
model = RandomForestClassifier()
model.fit(X, y)

# Get feature importance scores
importances = model.feature_importances_
print("Feature Importances:", importances)
```

(c) **Ensemble Feature Selection** Ensemble feature selection combines multiple selection techniques to enhance reliability. We gather a ton of data when training a model to improve machine learning. Not all of these data will be useful in creating the model, though. It's possible that some classes or a certain set of data won't help the model perform well. Having too much irrelevant data can cause the model to slow since the model is learning from too many irrelevant features. EFST combines the outputs of multiple feature selection algorithms to improve the performance of machine learning models. Due to its capacity to incorporate the advantages of many feature selection algorithms, ensemble feature selection techniques are growing in popularity in machine learning. The objective of ensemble feature selection is to produce a set of features that can most effectively enhance the predictive ability or functionality of our model.

Examples:

- **Combining Chi-Square, Information Gain, and RFE (Like EFST framework)**
- **Voting or Ranking of Features Across Different Methods**

Example Code for Ensemble Feature Selection

```
from sklearn.feature_selection import mutual_info_classif
from sklearn.preprocessing import StandardScaler

# Compute feature scores using multiple methods
chi2_scores = chi2(X, y)[0]
info_gain_scores = mutual_info_classif(X, y)
rfe.fit(X, y)
rfe_ranking = rfe.ranking_
```

```
# Normalize and sum scores for ensemble selection
scaler = StandardScaler()
scores = scaler.fit_transform(
    np.array([chi2_scores, info_gain_scores, -rfe_ranking]).T
)

final_scores = scores.sum(axis=1)
selected_features = np.argsort(final_scores)[-3:] # Top 3 features

print("Selected Features (Ensemble):", selected_features)
```

REAL WORLD APPLICATION OF EFST

In this book, we will demonstrate an EFST named '3ConFA Framework: 3 Conditions for feature aggregation' proposed by Akazue and Clive (2023) in their article , cybershield: harnessing ensemble feature selection technique For robust distributed denial of service attacks detection . 3ConFA integrates features extracted using multi-filter-based feature ranking selection and a wrapper-based feature subset selection viz; (i) Chi-square (ii) Information Gain (iii) DT(Decision Tree) -RFE. (i)and (ii) are filter-based feature ranking techniques, (iii) is a wrapper-based feature subset selection method.

The EFST consists of the combined workings of the three feature selection algorithms. Only the most important features are selected using RFE (features having 1 as a rank value). We calculated the average of Information gain, and Chi-squared algorithms and used the values computed as the three threshold values, *h1,h2, and h3.* For a feature to be selected from the dataset using the EFST, **Elements of the 3ConFA Framework**

1. **Chi-square (Chi2)** x^2 : This method utilizes the test of independence to assess whether the feature f is independent of the target variable. It evaluates the association between

the presence or absence of a feature and the target variable. It calculates the chi-squared statistic for each feature and the target variable. The higher the value, the more relevant the feature with respect to the class C (target) .

$$x^2 = \sum \frac{(O_i - \in_i)}{\in_i} \quad Equation\ 1$$

Where

O_i = Observed frequency of feature-class co-occurrence

$\in_i$ = Expected frequency (assuming no association between feature and class)

Sum ($\sum$) = Calculated across all categories of the feature and class

2. **Information Gain (IG)**

Information Gain measures how much a feature reduces uncertainty (entropy) about the target variable. It evaluates the importance of a feature by quantifying how well it splits the data into homogeneous groups. This approach offers an ordered ranking of each feature, and a threshold is then required.

Information Gain is found by taking the difference between a dataset's overall Entropy and its Conditional Entropy when considering a specific feature.

Entropy Calculation:

Measures impurity/disorder in the target variable C:

$$H(\mathrm{C}) = -\sum_i P(\mathrm{C}_i) \log_2 P(\mathrm{C}_i) \quad Equation\ 2$$

Conditional Entropy:

Computes entropy after splitting data by feature *f*:

$$H\left(\mathrm{C}\middle/f\right) = -\sum_{j} P(f_i) \sum_{i} P(C_i/f_j) \log_2 P(C_i/f_j) - - - - - - - - - - \quad Equation\ 3$$

IG Formula:

Difference between original entropy and post-split entropy:

$$IG(f) = H(\mathrm{C}) - H\left({}^{C}/_{F}\right) - - - - - - - - - - - - \quad Equation\ 4$$

3. **Decision Tree-Based Recursive Feature Elimination (DT-RFE):**

The less important features are removed by the method one by one, according to their importance scores in a decision tree. After that, the algorithm reduces the number of features, retrains the tree model and evaluates the feature scores again. This process keeps happening until an ideal set of features is found, balancing the effectiveness of the model and the simplicity of the features used. Unlike filter methods that ignore feature relationships, DT-RFE pays attention to connections between features to provide a detailed approach for selecting the best group of features as it repeatedly improves the performance of the model.

In short, The DT-RFE method keeps reducing the number of important features until there are as many features as desired. It continues training the classifier on the set of current features and removes those that play a minor role as indicated by the weights in the tree classifier.

4. ***Iff* Conditions**

The three different aspects of aggregating features are what define the 3ConFA framework. For a feature to be included, it has to meet three different requirements (requirements for aggregation).

Condition 1 (INFORMATION GAIN THRESHOLD): The score for information gain > = h1

$$IG \geq h1 \quad \text{Equation } 6$$

This is the information gain threshold. The condition means that the IG score of a feature must be at least as high as the mean IG score (h_1) among all features to be retained. It reveals how much uncertainty about the target is lessened when that feature is considered. This means that the features chosen are above the average for predictive relevance and are therefore not wasted.

It allows the system to reduce distortion by learning from new data. Keeping in mind, if the average IG (h_1) is 0.3, we would keep a feature with an IG of 0.4 like "transaction amount," but discount "user ID" with an IG of 0.1.

Condition 2 CHI-SQUARE THRESHOLD: The score for χ^2 > = h2

$$\boldsymbol{X^2 \geq h2} \quad \textbf{Equation 7}$$

The Chi-square threshold condition serves as a statistical filter to identify features that exhibit a meaningful association with the target variable. Here, the algorithm examines the Chi-square (χ^2) for every feature; the score indicates the strength of the link between the feature and the target class. The χ^2 score of a feature must be not less than h_2 which is usually defined by taking the average value of all features in the dataset. As a result of this rule, the method eliminates those features that fail to help in prediction and only keeps those that predict well.

Condition 3 DT-RFE THRESHOLD: DT-RFE Value = = 1

$$\boldsymbol{DT - RFE == 1} \quad \textbf{Equation 8}$$

Condition 3 specifies that for a feature to be selected from the dataset, its **Decision Tree-Recursive Feature Elimination (DT-**

RFE) score must be exactly 1. As a result of this criterion, only important features are chosen for training the model.

Using DT-RFE, the method removes features that make less of a difference to the model based on the evaluation of a decision tree classifier. A score of 1 in DT-RFE indicates that the feature is vital for the algorithm. Since the feature plays a major role in influencing the accuracy of the predictions, it cannot be removed. If the threshold is 1, any feature with less significance is removed during the selection process.

A feature is chosen by 3ConFA for the model *iff* the following is true*:*

$$Condition\ 1 == TRUE$$
$$Condition\ 2 == TRUE$$
$$Condition\ 3 == TRUE$$

Table 18-1: Conditional table for feature selection

Condition	True	False
Condition 1	✓	✗
Condition 2	✓	✗
Condition 3	✓	✗

Algorithm for the proposed 3ConFA Framework:

Input:

- *Dataset D with n features* $\{f_1, f_2, ..., f_n\}$
- *Feature selection measures: Information Gain, Chi-Squared, RFE*
- *Base estimator (Decision Tree by default)*
- *Threshold parameters:* α, β *(for aggregation conditions)*

Output:

- *Optimal feature subset X*
- *Model performance metrics*

Procedure:

1. Initialization:
   - X ← ∅ (empty set for final selected features)
   - S_1, S_2, S_3 ← ∅ (temporary sets for each method's results)
2. Mutual Information Filtering:
   - For each feature f in D:
     - Calculate MI score: $MI(f_i) = I(f_i; target)$
   - Compute mean MI score: $h_1 = (\sum MI(f_i))/n$
   - $S_1 \leftarrow \{f_i \mid MI(f_i) \geq \alpha \cdot h_1\}$ (features above threshold)
3. Chi-Squared Filtering:
   - For each feature f_i in D:
     - Calculate χ^2 score: $x^2(f_i) = \sum \frac{(O_i - \epsilon_i)}{\epsilon_i}$
   - Compute mean χ^2 score: $h_2 = (\sum \chi^2(f_i))/n$
   - $S_2 \leftarrow \{f_i \mid \chi^2(f_i) \geq \beta \cdot h_2\}$ (features above threshold)
4. Recursive Feature Elimination:
   a. Initialize: F ← all features, model ← base estimator
   b. Repeat until stopping condition met:
      i. Train model on current feature set F
      ii. Get importance scores for all f ∈ F
      iii. Rank features by importance
      iv. Eliminate bottom k features (e.g., k = 1)
      v. Evaluate model performance
   c. S_3 ← optimal feature subset from RFE
5. Feature Aggregation:
   - For each feature f in D:
     - if (f ∈ S_1 AND f ∈ S_2 AND f ∈ S_3):
       - X ← X ∪ {f}

- Alternatively, use voting:
- $X \leftarrow \{f \mid f \text{ appears in at least 2 of } S_1, S_2, S_3\}$

Example:

Consider the dataset below containing 20 features and their Chi-Square, RFE and IG values.

Feature Name	Chi-square Value	RFE (Rank)	Information Gain
Feature_1	15.23	1	0.45
Feature_2	12.75	1	0.38
Feature_3	18.90	1	0.52
Feature_4	10.34	5	0.33
Feature_5	20.11	1	0.56
Feature_6	9.78	8	0.29
Feature_7	14.56	6	0.41
Feature_8	7.89	10	0.22
Feature_9	19.32	1	0.49
Feature_10	8.65	12	0.25
Feature_11	16.48	1	0.44
Feature_12	13.21	11	0.37
Feature_13	21.09	1	0.60
Feature_14	6.43	16	0.19
Feature_15	11.57	1	0.31
Feature_16	5.98	18	0.17
Feature_17	17.64	13	0.47
Feature_18	4.76	20	0.12
Feature_19	22.30	17	0.62
Feature_20	3.89	19	0.10

Feature Name	Chi-square Value	RFE (Rank)	Information Gain
Mean	12.39	-	0.36

Recall:

Condition 1: *The score for information gain > = h1*

IG > =0.36

Condition 2: *The score for Chi-square> = h2*

Chi-sqaure> =12.39

Condition 3: *The RFE value = h3=1*

RFE=1

Applying the given conditions:

- Information Gain (IG) ≥ 0.36
- Chi-square Value ≥ 12.39
- RFE Rank = 1

Feature Name	Chi-square Value	RFE (Rank)	Information Gain
Feature_1	15.23	1	0.45
Feature_2	12.75	1	0.38
Feature_3	18.90	1	0.52
Feature_5	20.11	1	0.56
Feature_9	19.32	1	0.49
Feature_11	16.48	1	0.44
Feature_13	21.09	1	0.60

The EFST has successfully reduced the numbers of features required for training the model.

CASE STUDIES

We will provide real-world case studies to demonstrate the application of data science techniques.

Case Study 1: Predictive Analytics

Problem: Predict house prices based on features like location, size, and number of rooms.

Solution:

1. **Data Collection**: Use the Boston Housing dataset.
2. **Data Cleaning**: Handle missing values and outliers.
3. **Model Building**: Train a linear regression model.
4. **Evaluation**: Evaluate the model using mean squared error (MSE).

```
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load the dataset
data = load_boston()
X = data.data
y = data.target

# Train a model
model = LinearRegression()
model.fit(X, y)

# Make predictions
y_pred = model.predict(X)

# Evaluate the model
mse = mean_squared_error(y, y_pred)
print(f"Mean Squared Error: {mse}")
```

Case Study 2: Customer Segmentation

Problem: Group customers based on purchasing behavior to target marketing campaigns.

Solution:

1. **Data Collection**: Use a customer transaction dataset.
2. **Data Cleaning**: Normalize the data.
3. **Model Building**: Apply K-Means clustering.
4. **Evaluation**: Visualize the clusters.

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Sample data
X = [[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]]

# Train a model
model = KMeans(n_clusters=2)
model.fit(X)

# Visualize the clusters
plt.scatter([x[0] for x in X], [x[1] for x in X], c=model.labels_)
plt.show()
```

Case Study 3: Sentiment Analysis

Problem: Analyze customer reviews to determine sentiment (positive, negative, or neutral).

Solution:

1. **Data Collection**: Scrape reviews from a website or use a pre-existing dataset.
2. **Data Cleaning**: Remove stopwords and perform tokenization.
3. **Model Building**: Train a logistic regression model.
4. **Evaluation**: Evaluate the model using accuracy.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Sample data
reviews = ["I love this product!", "This is the worst product ever.", "It's okay."]
labels = [1, 0, 0] # 1 = Positive, 0 = Negative

# Convert text to features
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(reviews)

# Train a model
model = LogisticRegression()
model.fit(X, labels)

# Make predictions
y_pred = model.predict(X)

# Evaluate the model
accuracy = accuracy_score(labels, y_pred)
print(f"Accuracy: {accuracy}")
```

QUESTIONS

1. Define a problem statement for a data science project and identify the key success metrics.
2. Collect and clean a dataset for your project.
3. Build and evaluate a machine learning model using the cleaned dataset.
4. What are the key challenges in deploying a machine learning model?
5. Perform predictive analytics on a dataset of your choice and evaluate the model.
6. Use K-Means clustering to segment customers in a retail dataset.
7. Build a sentiment analysis model using customer reviews and evaluate its performance.
8. What are the key challenges in sentiment analysis?
9. Write a Python script to visualize the results of a clustering algorithm.

MODULE 19

HANDS-ON PROJECTS FOR DATA SCIENCE FUNDAMENTALS AND BEYOND

This module presents a set of projects that help you practice different types of data science. These assignments help students practice by working on real-life issues, keep learning key points and sharpen their skills in data, visualization, machine learning, deep learning and natural language processing with Python and many useful libraries.

At the beginning, the module includes simpler projects, like EDA, Regression and Sentiment Analysis using TextBlob. With these, you can start working on data in data science by loading data, preprocessing it, using basic models, visualizing the outcomes and understanding what has happened.

As you move forward, the projects involve more complicated work on topics such as predicting timelines, classifying images and text using deep learning, offering recommendations, detecting anomalies and making interactive dashboards. For example, predictive maintenance, fraud detection and market basket analysis represent data science in many industries.

All projects have the following things included in them:

- A clear objective
- Detailed requirements
- Practical use of libraries such as Pandas, Scikit-learn, Matplotlib, Seaborn, TextBlob, XGBoost, TensorFlow, Keras, NLTK, Plotly Dash, and Streamlit

- Opportunities for critical thinking and data-driven decision-making
- Emphasis on interpretation, not just implementation

If you are learning data science, aiming to become a data scientist or need to improve your skills, these projects help with learning step by step and using what you learn in practice. When this module is finished, you'll be able to highlight numerous data science projects that display your skills.

PROJECTS

Project 1: Perform EDA on a Simple Dataset
Objective: Perform simple analysis on a basic dataset (such as the Iris or Titanic dataset) as part of the objective.
Requirements:

- Load the dataset using Pandas.
- Display basic statistics (mean, median, mode).
- Visualize data using histograms, bar plots, and box plots.
- Interpret the distributions and relationships in the data.

Project 2: Basic Linear Regression for House Price Prediction
Objective: Forecast house prices by using the amount of the square footage or number of rooms in linear regression.
Requirements:

- Use a CSV with only house prices and some important features.
- Use Scikit-learn to implement linear regression.
- Imagine how the predicted prices compare to the actual ones.
- Assess the model's accuracy using its MSE.

Project 3: Simple Classification with Logistic Regression
Objective: Classify data into two categories (e.g., predicting whether a passenger survived on the Titanic based on features).
Requirements:

- Use a simple dataset like Titanic or a CSV file with binary labels.
- Preprocess data by handling missing values and encoding categorical variables.
- Train a logistic regression model using Scikit-learn.
- Evaluate the model using accuracy and confusion matrix.

Project 4: Basic Sentiment Analysis with TextBlob

Objective: Perform sentiment analysis on a set of text data (e.g., product reviews, tweets).

Requirements:

- Use the TextBlob library for sentiment analysis.
- Analyze the polarity and subjectivity of the text.
- Visualize the sentiment scores in a bar or pie chart.
- Interpret the overall sentiment of the dataset.

Project 5: Data Visualization of Weather Data

Objective: Visualize weather data (e.g., temperature, humidity, precipitation) over time.

Requirements:

- Use a simple weather dataset (e.g., from a CSV file or Kaggle).
- Create line plots for temperature over time, bar plots for monthly precipitation, etc.
- Use Matplotlib or Seaborn for visualizations.
- Add labels, legends, and titles to the plots.

Project 6: Real-World Data Exploration and Visualization

Objective: Using a publicly available dataset (e.g., Titanic, Iris, COVID-19, or World Bank), create a set of visualizations that explore key trends, relationships, and distributions in the data.

Requirements:

- Use at least three different types of plots (e.g., bar chart, heatmap, boxplot).
- Include one multi-panel visualization using Matplotlib or Seaborn.
- Add proper titles, labels, legends, and styling.
- Interpret at least one plot with a short paragraph.

Project 7: Build an Interactive Dashboard

Objective: Create an interactive data dashboard using Plotly Dash or Streamlit.

Requirements:

- Upload a dataset and allow user interaction (e.g., dropdowns, sliders).
- Include at least two dynamic charts.
- Customize layout and include text components for insight narration.
- Host locally and demonstrate usage.

Project 8: Time Series Forecasting with ARIMA

Objective: Forecast stock prices or demand for a product using historical time series data.

Requirements:

- Use ARIMA or SARIMA models for time series forecasting.
- Evaluate the model using metrics like RMSE or MAE.
- Visualize actual vs predicted values over time.
- Use residual analysis to validate the model.

Project 9: Deep Learning for Image Classification

Objective: Build a deep learning model to classify images from a dataset like CIFAR-10 or MNIST.

Requirements:

- Use CNN architectures such as ResNet, VGG, or MobileNet.
- Apply data augmentation techniques.
- Evaluate model performance using accuracy, F1 score, and confusion matrix.
- Use a pretrained model for transfer learning and compare performance.

Project 10: Natural Language Processing for Sentiment Analysis
Objective: Perform sentiment analysis on text data from a dataset like IMDB reviews or Twitter data.
Requirements:

- Use pre-trained word embeddings like Word2Vec or GloVe.
- Train a deep learning model (e.g., LSTM, GRU, or BERT).
- Implement text preprocessing (tokenization, stemming, lemmatization).
- Visualize sentiment distribution over time or by category.

Project 11: Predictive Maintenance for Industrial Equipment
Objective: Predict when industrial equipment will fail using sensor data (e.g., from turbines or motors).
Requirements:

- Implement XGBoost, Random Forest, or SVM for classification.
- Preprocess time series sensor data and engineer features.
- Use precision, recall, and F1 score for evaluation.
- Visualize sensor data patterns and failure trends.

Project 12: Fraud Detection in Financial Transactions
Objective: Build a machine learning model to detect fraudulent transactions in a financial dataset.
Requirements:

- Use techniques like XGBoost or Isolation Forest for anomaly detection.
- Handle class imbalance using SMOTE or other resampling techniques.
- Visualize feature importance and correlation matrices.
- Evaluate using precision, recall, and AUC-ROC curve.

Project 13: Recommender System for Movie Recommendations
Objective: Build a collaborative filtering or content-based recommender system using movie ratings data (e.g., MovieLens dataset).
Requirements:

- Implement collaborative filtering using matrix factorization or nearest neighbor algorithms.
- Evaluate the system's performance using RMSE or MAE.
- Visualize user-item matrix and recommendations.
- Implement model evaluation via cross-validation.

Project 14: Machine Translation with Neural Networks
Objective: Build a machine translation model to translate text from one language to another (e.g., English to French).
Requirements:

- Use Seq2Seq architecture with attention mechanism.
- Preprocess text data for tokenization and padding.
- Evaluate BLEU score for translation quality.
- Visualize translation accuracy for various phrases.

Project 15: Anomaly Detection in Cybersecurity
Objective: Detect network intrusions or fraudulent activity using network traffic data.
Requirements:

- Apply anomaly detection techniques (e.g., Isolation Forest, Autoencoders).
- Use feature engineering for network traffic data.
- Visualize anomalies detected in the traffic.
- Evaluate model performance using confusion matrix and ROC curve.

Project 16: Predicting Disease Spread with Machine Learning
Objective: Predict the spread of a disease (e.g., COVID-19) using epidemiological data.
Requirements:

- Use SIR (Susceptible, Infected, Recovered) model or machine learning techniques.
- Preprocess and clean time-series data.
- Visualize disease spread over time.
- Evaluate using mean absolute error or R-squared.

Project 17: Customer Segmentation with K-means Clustering
Objective: Perform customer segmentation for a retail company based on purchasing behavior.
Requirements:

- Apply K-means clustering algorithm for unsupervised learning.
- Use PCA for dimensionality reduction and data visualization.
- Evaluate the number of clusters using the elbow method.
- Visualize clusters and interpret customer profiles.

Project 18: Genetic Algorithm for Optimization Problems
Objective: Solve a complex optimization problem (e.g., traveling salesman, scheduling) using a genetic algorithm.
Requirements:

- Implement a genetic algorithm to find the optimal solution.
- Visualize the evolution of the population.
- Compare results with traditional optimization techniques.
- Analyze the convergence of the algorithm.

Project 19: Predicting Loan Default with Gradient Boosting

Objective: Predict loan default using financial and demographic data.

Requirements:

- Implement Gradient Boosting or XGBoost for classification.
- Handle missing data and imbalanced classes.
- Visualize important features affecting loan default.
- Evaluate using confusion matrix, AUC-ROC, and precision-recall curves.

Project 20: Market Basket Analysis with Association Rule Learning

Objective: Perform market basket analysis on transactional data using the Apriori algorithm.

Requirements:

- Implement the Apriori algorithm for frequent itemset mining.
- Visualize association rules with lift and confidence metrics.
- Identify key product relationships and trends.
- Interpret the discovered rules for actionable business insights.

Project 21: Image Captioning with Deep Learning

Objective: Generate captions for images using a deep learning model.

Requirements:

- Combine CNN for image feature extraction and RNN for caption generation.
- Use a dataset like MSCOCO for image-caption pairs.
- Evaluate model performance using BLEU score or CIDEr score.
- Visualize images with their generated captions.

Project 22: Text Generation with Recurrent Neural Networks
Objective: Generate text based on a corpus of literature or a specific theme.
Requirements:

- Implement LSTM or GRU-based models for text generation.
- Preprocess text data for training (tokenization, padding).
- Evaluate the quality of generated text.
- Visualize the training loss over epochs.

Project 23: Neural Style Transfer for Image Transformation
Objective: Apply neural style transfer to create artistic versions of images.
Requirements:

- Implement a deep neural network to extract content and style representations.
- Combine content and style representations to generate new images.
- Visualize the transformation results with original and stylized images.
- Experiment with different styles (e.g., Van Gogh, Picasso).

Project 24: Predicting Housing Prices with Ensemble Methods
Objective: Predict housing prices using features like location, size, and number of rooms.
Requirements:

- Implement ensemble techniques such as Random Forest and Gradient Boosting.
- Handle categorical and numerical features.
- Visualize important features affecting house prices.
- Evaluate model performance using RMSE or R-squared.

Project 25: Building a Chatbot with NLP

Objective: Build a conversational AI chatbot that can interact with users in natural language.

Requirements:

- Implement an NLP model (e.g., Seq2Seq with attention).
- Preprocess text data for tokenization and padding.
- Evaluate chatbot performance using response accuracy or BLEU score.
- Visualize chatbot interactions over time.

Project 26: Image Super-Resolution with Deep Learning

Objective: Improve the resolution of images using deep learning techniques.

Requirements:

- Use techniques like SRCNN or GANs for image enhancement.
- Train a deep learning model on a dataset like DIV2K.
- Visualize original and enhanced images for comparison.
- Evaluate image quality using PSNR or SSIM metrics.

Project 27: Social Media Sentiment and Trend Analysis

Objective: Analyze social media data (e.g., Twitter) for sentiment trends over time.

Requirements:

- Scrape or use an API to collect social media data.
- Perform sentiment analysis using a pre-trained model like VADER or BERT.
- Visualize sentiment trends over time (e.g., positive, neutral, negative).
- Identify trending topics using text mining techniques (e.g., TF-IDF).

Project 28: Real-Time Multi-Modal Emotion Recognition System

Objective: Build a real-time system that detects human emotions using both facial expressions (video) and speech signals.

Requirements:

- Collect or use a dataset combining facial expression videos and corresponding audio (e.g., RAVDESS, CREMA-D).
- Use deep learning for feature extraction (e.g., CNNs for video frames, RNNs or transformers for audio signals).
- Fuse multi-modal features using an attention mechanism or late fusion strategy.
- Build a real-time pipeline using OpenCV and PyAudio.
- Deploy the model using TensorFlow Serving or TorchServe.
- Evaluate using metrics like accuracy, F1-score, and latency.

Project 29: End-to-End AutoML Pipeline with Neural Architecture Search

Objective: Implement a custom AutoML system that performs feature engineering, model selection, and neural architecture search (NAS).

Requirements:

- Build a data preprocessing pipeline with feature type inference, encoding, scaling, and imputation.
- Implement a neural architecture search algorithm (e.g., Reinforcement Learning-based or Differentiable NAS).
- Automate hyperparameter optimization with Bayesian Optimization or Tree-structured Parzen Estimators (TPE).
- Evaluate on multiple datasets and track model performance using MLflow or Weights & Biases.
- Provide a CLI or web interface to upload a dataset and run full AutoML.

- Optimize for both accuracy and computational efficiency (multi-objective optimization).

Project 30: Distributed Deep Learning on a Big Dataset with Spark and Horovod

Objective: Train a large-scale deep learning model on distributed clusters using Apache Spark and Horovod.

Requirements:

- Use a big dataset (e.g., Open Images, YouTube-8M, or large tabular data from Kaggle).
- Set up a distributed computing environment using Apache Spark on AWS EMR, GCP, or a local cluster.
- Integrate Horovod with TensorFlow or PyTorch for distributed training.
- Implement efficient data loading pipelines (e.g., TFRecord, HDFS).
- Apply checkpointing, early stopping, and fault-tolerance mechanisms.
- Benchmark the performance gains vs single-machine training.

REFRENCES

Morgan, P. (2016). *Data analysis from scratch with Python: Step-by-step guide*. AI Sciences LLC.
ISBN-13: 978-1721942817
ISBN-10: 1721942815

Igual, L., & Segui, S. (2017). *Introduction to data science: A Python approach to concepts, techniques, and applications*. Springer.
https://doi.org/10.1007/978-3-319-50017-1
ISBN 978-3-319-50016-4
ISBN 978-3-319-50017-1 (eBook)
ISSN 1863-7310
ISSN 2197-1781 (electronic)
Library of Congress Control Number: 2016962046

McKinney, W. (2018). *Python for data analysis* (2nd ed.). O'Reilly Media.